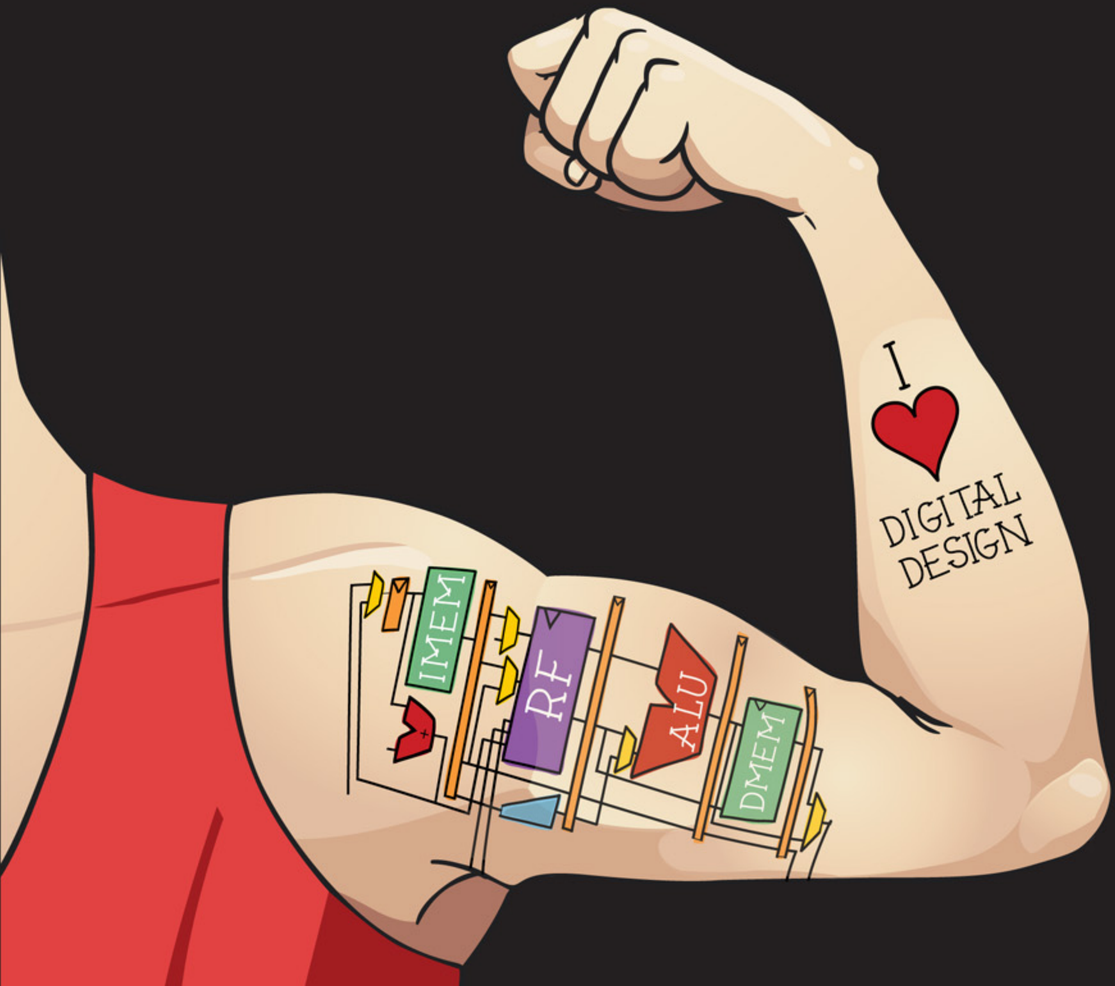
Digital Design and Computer Architecture

ARM® Edition


IMEM
RF
ALU
DMEM
I
DIGITAL DESIGN

MORGAN KAUFMANN

Sarah L. Harris & David Money Harris

In Praise of *Digital Design and Computer Architecture*

ARM® Edition

Harris and Harris have done a remarkable and commendable job in creating a true textbook which clearly shows their love and passion for teaching and educating. The students who read this book will be thankful to Harris and Harris for many years after graduation. The writing style, the clearness, the detailed diagrams, the flow of information, the gradual increase in the complexity of the subjects, the great examples throughout the chapters, the exercises at the end of the chapters, the concise yet clear explanations, the useful real-world examples, the coverage of all aspects of each topic—all of these things are done very well. If you are a student using this book for your course get ready to have fun, be impressed, and learn a great deal as well!

Mehdi Hatamian, Sr. Vice President, Broadcom

Harris and Harris have done an excellent job creating this ARM version of their popular book, *Digital Design and Computer Architecture*. Retargeting to ARM is a challenging task, but the authors have done it successfully while maintaining their clear and thorough presentation style, as well as their outstanding documentation quality. I believe this new edition will be very much welcomed by both students and professionals.

Donald Hung, San Jose State University

Of all the textbooks I've reviewed and assigned in my 10 years as a professor, *Digital Design and Computer Architecture* is one of only two that is unquestionably worth buying. (The other is *Computer Organization and Design*.) The writing is clear and concise; the diagrams are easy to understand; and the CPU the authors use as a running example is complex enough to be realistic, yet simple enough to be thoroughly understood by my students.

Zachary Kurmas, Grand Valley State University

Digital Design and Computer Architecture brings a fresh perspective to an old discipline. Many textbooks tend to resemble overgrown shrubs, but Harris and Harris have managed to prune away the deadwood while preserving the fundamentals and presenting them in a contemporary context. In doing so, they offer a text that will benefit students interested in designing solutions for tomorrow's challenges.

Jim Frenzel, University of Idaho

Harris and Harris have a pleasant and informative writing style. Their treatment of the material is at a good level for introducing students to computer engineering with plenty of helpful diagrams. Combinational circuits, microarchitecture, and memory systems are handled particularly well.

James Pinter-Lucke, Claremont McKenna College

Harris and Harris have written a book that is very clear and easy to understand. The exercises are well-designed and the real-world examples are a nice touch. The lengthy and confusing explanations often found in similar textbooks are not seen here. It's obvious that the authors have devoted a great deal of time and effort to create an accessible text. I strongly recommend *Digital Design and Computer Architecture*.

Peiyi Zhao, Chapman University

Digital Design and Computer Architecture

ARM® Edition

This page intentionally left blank

Digital Design and Computer Architecture

ARM® Edition

Sarah L. Harris
David Money Harris

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK
MORGAN KAUFMANN

Acquiring Editor: Steve Merken
Development Editor: Nate McFadden
Project Manager: Punithavathy Govindaradjane
Designer: Vicky Pearson

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

Copyright © 2016 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

All material relating to ARM® technology has been reproduced with permission from ARM Limited, and should only be used for education purposes. All ARM -based models shown or referred to in the text must not be used, reproduced or distributed for commercial purposes, and in no event shall purchasing this textbook be construed as granting you or any third party, expressly or by implication, estoppel or otherwise, a license to use any other ARM technology or know how. Materials provided by ARM are copyright © ARM Limited (or its affiliates).

ISBN: 978-0-12-800056-4

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

For Information on all Morgan Kaufmann publications,
visit our website at www.mkp.com

Printed and bound in the United States of America

ELSEVIER

Book Aid
International

Working together
to grow libraries in
developing countries

www.elsevier.com • www.bookaid.org

To our families

This page intentionally left blank

Contents

Preface xix

Features xx
Online Supplements xxi
How to Use the Software Tools in a Course xxii
Labs xxii
Bugs xxiii
Acknowledgments xxiv

Chapter 1 From Zero to One 3

1.1 The Game Plan 3
1.2 The Art of Managing Complexity 4
  1.2.1 *Abstraction* 4
  1.2.2 *Discipline* 5
  1.2.3 *The Three-Y's* 6
1.3 The Digital Abstraction 7
1.4 Number Systems 9
  1.4.1 *Decimal Numbers* 9
  1.4.2 *Binary Numbers* 9
  1.4.3 *Hexadecimal Numbers* 11
  1.4.4 *Bytes, Nibbles, and All That Jazz* 13
  1.4.5 *Binary Addition* 14
  1.4.6 *Signed Binary Numbers* 15
1.5 Logic Gates 19
  1.5.1 *NOT Gate* 20
  1.5.2 *Buffer* 20
  1.5.3 *AND Gate* 20
  1.5.4 *OR Gate* 21
  1.5.5 *Other Two-Input Gates* 21
  1.5.6 *Multiple-Input Gates* 21
1.6 Beneath the Digital Abstraction 22
  1.6.1 *Supply Voltage* 22
  1.6.2 *Logic Levels* 22
  1.6.3 *Noise Margins* 23
  1.6.4 *DC Transfer Characteristics* 24
  1.6.5 *The Static Discipline* 24

1.7 CMOS Transistors 26
  1.7.1 *Semiconductors* 27
  1.7.2 *Diodes* 27
  1.7.3 *Capacitors* 28
  1.7.4 *nMOS and pMOS Transistors* 28
  1.7.5 *CMOS NOT Gate* 31
  1.7.6 *Other CMOS Logic Gates* 31
  1.7.7 *Transmission Gates* 33
  1.7.8 *Pseudo-nMOS Logic* 33
1.8 Power Consumption 34
1.9 Summary and a Look Ahead 35
Exercises 37
Interview Questions 52

Chapter 2 Combinational Logic Design 55

2.1 Introduction 55
2.2 Boolean Equations 58
  2.2.1 *Terminology* 58
  2.2.2 *Sum-of-Products Form* 58
  2.2.3 *Product-of-Sums Form* 60
2.3 Boolean Algebra 60
  2.3.1 *Axioms* 61
  2.3.2 *Theorems of One Variable* 61
  2.3.3 *Theorems of Several Variables* 62
  2.3.4 *The Truth Behind It All* 64
  2.3.5 *Simplifying Equations* 65
2.4 From Logic to Gates 66
2.5 Multilevel Combinational Logic 69
  2.5.1 *Hardware Reduction* 70
  2.5.2 *Bubble Pushing* 71
2.6 X's and Z's, Oh My 73
  2.6.1 *Illegal Value: X* 73
  2.6.2 *Floating Value: Z* 74
2.7 Karnaugh Maps 75
  2.7.1 *Circular Thinking* 76
  2.7.2 *Logic Minimization with K-Maps* 77
  2.7.3 *Don't Cares* 81
  2.7.4 *The Big Picture* 82
2.8 Combinational Building Blocks 83
  2.8.1 *Multiplexers* 83
  2.8.2 *Decoders* 86
2.9 Timing 88
  2.9.1 *Propagation and Contamination Delay* 88
  2.9.2 *Glitches* 92

2.10 Summary 95
Exercises 97
Interview Questions 106

Chapter 3 Sequential Logic Design 109

3.1 Introduction 109
3.2 Latches and Flip-Flops 109
  3.2.1 *SR Latch* 111
  3.2.2 *D Latch* 113
  3.2.3 *D FIip-Flop* 114
  3.2.4 *Register* 114
  3.2.5 *Enabled Flip-Flop* 115
  3.2.6 *Resettable Flip-Flop* 116
  3.2.7 *Transistor-Level Latch and Flip-Flop Designs* 116
  3.2.8 *Putting It All Together* 118
3.3 Synchronous Logic Design 119
  3.3.1 *Some Problematic Circuits* 119
  3.3.2 *Synchronous Sequential Circuits* 120
  3.3.3 *Synchronous and Asynchronous Circuits* 122
3.4 Finite State Machines 123
  3.4.1 *FSM Design Example* 123
  3.4.2 *State Encodings* 129
  3.4.3 *Moore and Mealy Machines* 132
  3.4.4 *Factoring State Machines* 134
  3.4.5 *Deriving an FSM from a Schematic* 137
  3.4.6 *FSM Review* 140
3.5 Timing of Sequential Logic 141
  3.5.1 *The Dynamic Discipline* 142
  3.5.2 *System Timing* 142
  3.5.3 *Clock Skew* 148
  3.5.4 *Metastability* 151
  3.5.5 *Synchronizers* 152
  3.5.6 *Derivation of Resolution Time* 154
3.6 Parallelism 157
3.7 Summary 161
Exercises 162
Interview Questions 171

Chapter 4 Hardware Description Languages 173

4.1 Introduction 173
  4.1.1 *Modules* 173
  4.1.2 *Language Origins* 174
  4.1.3 *Simulation and Synthesis* 175

4.2 Combinational Logic 177
  4.2.1 *Bitwise Operators* 177
  4.2.2 *Comments and White Space* 180
  4.2.3 *Reduction Operators* 180
  4.2.4 *Conditional Assignment* 181
  4.2.5 *Internal Variables* 182
  4.2.6 *Precedence* 184
  4.2.7 *Numbers* 185
  4.2.8 *Z's and X's* 186
  4.2.9 *Bit Swizzling* 188
  4.2.10 *Delays* 188
4.3 Structural Modeling 190
4.4 Sequential Logic 193
  4.4.1 *Registers* 193
  4.4.2 *Resettable Registers* 194
  4.4.3 *Enabled Registers* 196
  4.4.4 *Multiple Registers* 197
  4.4.5 *Latches* 198
4.5 More Combinational Logic 198
  4.5.1 *Case Statements* 201
  4.5.2 *If Statements* 202
  4.5.3 *Truth Tables with Don't Cares* 205
  4.5.4 *Blocking and Nonblocking Assignments* 205
4.6 Finite State Machines 209
4.7 Data Types 213
  4.7.1 *SystemVerilog* 214
  4.7.2 *VHDL* 215
4.8 Parameterized Modules 217
4.9 Testbenches 220
4.10 Summary 224
Exercises 226
Interview Questions 237

Chapter 5 Digital Building Blocks 239

5.1 Introduction 239
5.2 Arithmetic Circuits 239
  5.2.1 *Addition* 239
  5.2.2 *Subtraction* 246
  5.2.3 *Comparators* 246
  5.2.4 *ALU* 248
  5.2.5 *Shifters and Rotators* 251
  5.2.6 *Multiplication* 252

5.2.7 *Division* 254
5.2.8 *Further Reading* 255
5.3 Number Systems 255
5.3.1 *Fixed-Point Number Systems* 255
5.3.2 *Floating-Point Number Systems* 256
5.4 Sequential Building Blocks 259
5.4.1 *Counters* 260
5.4.2 *Shift Registers* 261
5.5 Memory Arrays 264
5.5.1 *Overview* 264
5.5.2 *Dynamic Random Access Memory (DRAM)* 266
5.5.3 *Static Random Access Memory (SRAM)* 267
5.5.4 *Area and Delay* 267
5.5.5 *Register Files* 268
5.5.6 *Read Only Memory* 268
5.5.7 *Logic Using Memory Arrays* 270
5.5.8 *Memory HDL* 271
5.6 Logic Arrays 271
5.6.1 *Programmable Logic Array* 272
5.6.2 *Field Programmable Gate Array* 274
5.6.3 *Array Implementations* 279
5.7 Summary 281
Exercises 282
Interview Questions 293

Chapter 6 Architecture 295

6.1 Introduction 295
6.2 Assembly Language 296
6.2.1 *Instructions* 297
6.2.2 *Operands: Registers, Memory, and Constants* 298
6.3 Programming 303
6.3.1 *Data-processing Instructions* 303
6.3.2 *Condition Flags* 306
6.3.3 *Branching* 308
6.3.4 *Conditional Statements* 309
6.3.5 *Getting Loopy* 312
6.3.6 *Memory* 313
6.3.7 *Function Calls* 317
6.4 Machine Language 329
6.4.1 *Data-processing Instructions* 329
6.4.2 *Memory Instructions* 333
6.4.3 *Branch Instructions* 334
6.4.4 *Addressing Modes* 336

6.4.5 *Interpreting Machine Language Code* 336
6.4.6 *The Power of the Stored Program* 337
6.5 Lights, Camera, Action: Compiling, Assembling, and Loading 339
6.5.1 *The Memory Map* 339
6.5.2 *Compilation* 340
6.5.3 *Assembling* 342
6.5.4 *Linking* 343
6.5.5 *Loading* 344
6.6 Odds and Ends 345
6.6.1 *Loading Literals* 345
6.6.2 *NOP* 346
6.6.3 *Exceptions* 347
6.7 Evolution of ARM Architecture 350
6.7.1 *Thumb Instruction Set* 351
6.7.2 *DSP Instructions* 352
6.7.3 *Floating-Point Instructions* 357
6.7.4 *Power-Saving and Security Instructions* 358
6.7.5 *SIMD Instructions* 358
6.7.6 *64-bit Architecture* 360
6.8 Another Perspective: x86 Architecture 360
6.8.1 *x86 Registers* 362
6.8.2 *x86 Operands* 362
6.8.3 *Status Flags* 363
6.8.4 *x86 Instructions* 364
6.8.5 *x86 Instruction Encoding* 364
6.8.6 *Other x86 Peculiarities* 367
6.8.7 *The Big Picture* 368
6.9 Summary 368
Exercises 370
Interview Questions 383

Chapter 7 Microarchitecture 385

7.1 Introduction 385
7.1.1 *Architectural State and Instruction Set* 385
7.1.2 *Design Process* 386
7.1.3 *Microarchitectures* 388
7.2 Performance Analysis 389
7.3 Single-Cycle Processor 390
7.3.1 *Single-Cycle Datapath* 390
7.3.2 *Single-Cycle Control* 397
7.3.3 *More Instructions* 402
7.3.4 *Performance Analysis* 402

7.4 Multicycle Processor ... 406
  7.4.1 *Multicycle Datapath* ... 407
  7.4.2 *Multicycle Control* ... 413
  7.4.3 *Performance Analysis* ... 421
7.5 Pipelined Processor ... 425
  7.5.1 *Pipelined Datapath* ... 428
  7.5.2 *Pipelined Control* ... 430
  7.5.3 *Hazards* ... 431
  7.5.4 *Performance Analysis* ... 441
7.6 HDL Representation ... 443
  7.6.1 *Single-Cycle Processor* ... 444
  7.6.2 *Generic Building Blocks* ... 449
  7.6.3 *Testbench* ... 452
7.7 Advanced Microarchitecture ... 456
  7.7.1 *Deep Pipelines* ... 457
  7.7.2 *Micro-Operations* ... 458
  7.7.3 *Branch Prediction* ... 459
  7.7.4 *Superscalar Processor* ... 461
  7.7.5 *Out-of-Order Processor* ... 463
  7.7.6 *Register Renaming* ... 465
  7.7.7 *Multithreading* ... 467
  7.7.8 *Multiprocessors* ... 468
7.8 Real-World Perspective: Evolution of ARM Microarchitecture ... 470
7.9 Summary ... 476
Exercises ... 478
Interview Questions ... 484

Chapter 8 Memory Systems ... 487

8.1 Introduction ... 487
8.2 Memory System Performance Analysis ... 491
8.3 Caches ... 492
  8.3.1 *What Data is Held in the Cache?* ... 493
  8.3.2 *How is Data Found?* ... 494
  8.3.3 *What Data is Replaced?* ... 502
  8.3.4 *Advanced Cache Design* ... 503
  8.3.5 *The Evolution of ARM Caches* ... 507
8.4 Virtual Memory ... 508
  8.4.1 *Address Translation* ... 510
  8.4.2 *The Page Table* ... 512
  8.4.3 *The Translation Lookaside Buffer* ... 514
  8.4.4 *Memory Protection* ... 515
  8.4.5 *Replacement Policies* ... 516
  8.4.6 *Multilevel Page Tables* ... 516

8.5 Summary 518
Epilogue 519
Exercises 520
Interview Questions 529

Chapter 9 I/O Systems 531

9.1 Introduction 531

Chapter 9 is available as an online supplement 531.e1

9.1 Introduction 531.e1
9.2 Memory-Mapped I/O 531.e1
9.3 Embedded I/O Systems 531.e3
  9.3.1 *BCM2835 System-on-Chip* 531.e4
  9.3.2 *Device Drivers* 531.e6
  9.3.3 *General-Purpose Digital I/O* 531.e8
  9.3.4 *Serial I/O* 531.e11
  9.3.5 *Timers* 531.e23
  9.3.6 *Analog I/O* 531.e25
  9.3.7 *Interrupts* 531.e32
9.4 Other Microcontroller Peripherals 531.e32
  9.4.1 *Character LCDs* 531.e33
  9.4.2 *VGA Monitor* 531.e36
  9.4.3 *Bluetooth Wireless Communication* 531.e42
  9.4.4 *Motor Control* 531.e43
9.5 Bus Interfaces 531.e54
  9.5.1 *AHB-Lite* 531.e54
  9.5.2 *Memory and Peripheral Interface Example* 531.e55
9.6 PC I/O Systems 531.e57
  9.6.1 *USB* 531.e59
  9.6.2 *PCI and PCI Express* 531.e59
  9.6.3 *DDR3 Memory* 531.e60
  9.6.4 *Networking* 531.e61
  9.6.5 *SATA* 531.e61
  9.6.6 *Interfacing to a PC* 531.e62
9.7 Summary 531.e64

Appendix A Digital System Implementation 533

A.1 Introduction 533

Appendix A is available as an online supplement 533.e1

A.1 Introduction 533.e1

A.2 74xx Logic 533.e1
  A.2.1 *Logic Gates* 533.e2
  A.2.2 *Other Functions* 533.e2
A.3 Programmable Logic 533.e2
  A.3.1 *PROMs* 533.e2
  A.3.2 *PLAs* 533.e6
  A.3.3 *FPGAs* 533.e7
A.4 Application-Specific Integrated Circuits 533.e9
A.5 Data Sheets 533.e9
A.6 Logic Families 533.e15
A.7 Packaging and Assembly 533.e17
A.8 Transmission Lines 533.e20
  A.8.1 *Matched Termination* 533.e22
  A.8.2 *Open Termination* 533.e24
  A.8.3 *Short Termination* 533.e25
  A.8.4 *Mismatched Termination* 533.e25
  A.8.5 *When to Use Transmission Line Models* 533.e28
  A.8.6 *Proper Transmission Line Terminations* 533.e28
  A.8.7 *Derivation of* Z_0 .... 533.e30
  A.8.8 *Derivation of the Reflection Coefficient* 533.e31
  A.8.9 *Putting It All Together* 533.e32
A.9 Economics 533.e33

Appendix B ARM Instructions 535

B.1 Data-Processing Instructions 535
  B.1.1 *Multiply Instructions* 537
B.2 Memory Instructions 538
B.3 Branch Instructions 539
B.4 Miscellaneous Instructions 539
B.5 Condition Flags 540

Appendix C C Programming 541

C.1 Introduction 541

Appendix C is available as an online supplement 541.e1

C.1 Introduction 541.e1
C.2 Welcome to C 541.e3
  C.2.1 *C Program Dissection* 541.e3
  C.2.2 *Running a C Program* 541.e4
C.3 Compilation 541.e5
  C.3.1 *Comments* 541.e5
  C.3.2 `#define` 541.e5
  C.3.3 `#include` 541.e6

C.4 Variables 541.e7
  C.4.1 *Primitive Data Types* 541.e8
  C.4.2 *Global and Local Variables* 541.e9
  C.4.3 *Initializing Variables* 541.e11
C.5 Operators 541.e11
C.6 Function Calls 541.e15
C.7 Control-Flow Statements 541.e16
  C.7.1 *Conditional Statements* 541.e17
  C.7.2 *Loops* 541.e19
C.8 More Data Types 541.e21
  C.8.1 *Pointers* 541.e21
  C.8.2 *Arrays* 541.e23
  C.8.3 *Characters* 541.e27
  C.8.4 *Strings* 541.e28
  C.8.5 *Structures* 541.e29
  C.8.6 `typedef` 541.e31
  C.8.7 *Dynamic Memory Allocation* 541.e32
  C.8.8 *Linked Lists* 541.e33
C.9 Standard Libraries 541.e35
  C.9.1 *stdio* 541.e35
  C.9.2 *stdlib* 541.e40
  C.9.3 *math* 541.e42
  C.9.4 *string* 541.e43
C.10 Compiler and Command Line Options 541.e43
  C.10.1 *Compiling Multiple C Source Files* 541.e43
  C.10.2 *Compiler Options* 541.e43
  C.10.3 *Command Line Arguments* 541.e44
C.11 Common Mistakes 541.e45

Index 543

Preface

This book is unique in its treatment in that it presents digital logic design from the perspective of computer architecture, starting at the beginning with 1's and 0's, and leading through the design of a microprocessor.

We believe that building a microprocessor is a special rite of passage for engineering and computer science students. The inner workings of a processor seem almost magical to the uninitiated, yet prove to be straightforward when carefully explained. Digital design in itself is a powerful and exciting subject. Assembly language programming unveils the inner language spoken by the processor. Microarchitecture is the link that brings it all together.

The first two editions of this increasingly popular text have covered the MIPS architecture in the tradition of the widely used architecture books by Patterson and Hennessy. As one of the original Reduced Instruction Set Computing architectures, MIPS is clean and exceptionally easy to understand and build. MIPS remains an important architecture and has been infused with new energy after Imagination Technologies acquired it in 2013.

Over the past two decades, the ARM architecture has exploded in popularity because of its efficiency and rich ecosystem. More than 50 billion ARM processors have been shipped, and more than 75% of humans on the planet use products with ARM processors. At the time of this writing, nearly every cell phone and tablet sold contains one or more ARM processors. Forecasts predict tens of billions more ARM processors soon controlling the Internet of Things. Many companies are building high-performance ARM systems to challenge Intel in the server market. Because of the commercial importance and student interest, we have developed this ARM edition of this book.

Pedagogically, the learning objectives of the MIPS and ARM editions are identical. The ARM architecture has a number of features including addressing modes and conditional execution that contribute to its efficiency but add a small amount of complexity. The microarchitectures also are very similar, with conditional execution and the program counter being the largest changes. The chapter on I/O provides numerous examples using the Raspberry Pi, a very popular ARM-based embedded Linux single board computer.

We expect to offer both MIPS and ARM editions as long as the market demands.

FEATURES

Side-by-Side Coverage of SystemVerilog and VHDL

Hardware description languages (HDLs) are at the center of modern digital design practices. Unfortunately, designers are evenly split between the two dominant languages, SystemVerilog and VHDL. This book introduces HDLs in Chapter 4 as soon as combinational and sequential logic design has been covered. HDLs are then used in Chapters 5 and 7 to design larger building blocks and entire processors. Nevertheless, Chapter 4 can be skipped and the later chapters are still accessible for courses that choose not to cover HDLs.

This book is unique in its side-by-side presentation of SystemVerilog and VHDL, enabling the reader to learn the two languages. Chapter 4 describes principles that apply to both HDLs, and then provides language-specific syntax and examples in adjacent columns. This side-by-side treatment makes it easy for an instructor to choose either HDL, and for the reader to transition from one to the other, either in a class or in professional practice.

ARM Architecture and Microarchitecture

Chapters 6 and 7 offer the first in-depth coverage of the ARM architecture and microarchitecture. ARM is an ideal architecture because it is a real architecture shipped in millions of products yearly, yet it is streamlined and easy to learn. Moreover, because of its popularity in the commercial and hobbyist worlds, simulation and development tools exist for the ARM architecture. All material relating to ARM® technology has been reproduced with permission from ARM Limited.

Real-World Perspectives

In addition to the real-world perspective in discussing the ARM architecture, Chapter 6 illustrates the architecture of Intel x86 processors to offer another perspective. Chapter 9 (available as an online supplement) also describes peripherals in the context of the Raspberry Pi single-board computer, a hugely popular ARM-based platform. These real-world perspective chapters show how the concepts in the chapters relate to the chips found in many PCs and consumer electronics.

Accessible Overview of Advanced Microarchitecture

Chapter 7 includes an overview of modern high-performance microarchitectural features including branch prediction, superscalar, and out-of-order operation, multithreading, and multicore processors. The treatment is accessible to a student in a first course and shows how the microarchitectures in the book can be extended to modern processors.

End-of-Chapter Exercises and Interview Questions

The best way to learn digital design is to do it. Each chapter ends with numerous exercises to practice the material. The exercises are followed by a set of interview questions that our industrial colleagues have asked students who are applying for work in the field. These questions provide a helpful glimpse into the types of problems that job applicants will typically encounter during the interview process. Exercise solutions are available via the book's companion and instructor websites.

ONLINE SUPPLEMENTS

Supplementary materials are available online at http://textbooks.elsevier.com/9780128000564. This companion site (accessible to all readers) includes the following:

- Solutions to odd-numbered exercises
- Links to professional-strength computer-aided design (CAD) tools from Altera®
- Link to Keil's ARM Microcontroller Development Kit (MDK-ARM), a tool for compiling, assembling, and simulating C and assembly code for ARM processors
- Hardware description language (HDL) code for the ARM processor
- Altera Quartus II helpful hints
- Lecture slides in PowerPoint (PPT) format
- Sample course and laboratory materials
- List of errata

The instructor site (linked to the companion site and accessible to adopters who register at http://textbooks.elsevier.com/9780128000564) includes the following:

- Solutions to all exercises
- Links to professional-strength computer-aided design (CAD) tools from Altera®
- Figures from the text in PDF and PPT formats

Additional details on using the Altera, Raspberry Pi, and MDK-ARM tools in your course are provided. Details on the sample laboratory materials are also provided here.

HOW TO USE THE SOFTWARE TOOLS IN A COURSE

Altera Quartus II

Quartus II Web Edition is a free version of the professional-strength Quartus™ II FPGA design tools. It allows students to enter their digital designs in schematic or using either the SystemVerilog or the VHDL hardware description language (HDL). After entering the design, students can simulate their circuits using ModelSim™-Altera Starter Edition, which is available with the Altera Quartus II Web Edition. Quartus II Web Edition also includes a built-in logic synthesis tool supporting both SystemVerilog and VHDL.

The difference between Web Edition and Subscription Edition is that Web Edition supports a subset of the most common Altera FPGAs. The difference between ModelSim-Altera Starter Edition and ModelSim commercial versions is that the Starter Edition degrades performance for simulations with more than 10,000 lines of HDL.

Keil's ARM Microcontroller Development Kit (MDK-ARM)

Keil's MDK-ARM is a tool for developing code for an ARM processor. It is available for free download. The MDK-ARM includes a commercial ARM C compiler and a simulator that allows students to write both C and assembly programs, compile them, and then simulate them.

LABS

The companion site includes links to a series of labs that cover topics from digital design through computer architecture. The labs teach students how to use the Quartus II tools to enter, simulate, synthesize, and implement their designs. The labs also include topics on C and assembly language programming using the MDK-ARM and Raspberry Pi development tools.

After synthesis, students can implement their designs using the Altera DE2 (or DE2-115) Development and Education Board. This powerful and competitively priced board is available from www.altera.com. The board contains an FPGA that can be programmed to implement student designs. We provide labs that describe how to implement a selection of designs on the DE2 Board using Quartus II Web Edition.

To run the labs, students will need to download and install Altera Quartus II Web Edition and either MDK-ARM or the Raspberry Pi tools. Instructors may also choose to install the tools on lab machines. The labs include instructions on how to implement the projects on the DE2 Board. The implementation step may be skipped, but we have found it of great value.

We have tested the labs on Windows, but the tools are also available for Linux.

BUGS

As all experienced programmers know, any program of significant complexity undoubtedly contains bugs. So, too, do books. We have taken great care to find and squash the bugs in this book. However, some errors undoubtedly do remain. We will maintain a list of errata on the book's webpage.

Please send your bug reports to ddcabugs@gmail.com. The first person to report a substantive bug with a fix that we use in a future printing will be rewarded with a $1 bounty!

ACKNOWLEDGMENTS

We appreciate the hard work of Nate McFadden, Joe Hayton, Punithavathy Govindaradjane, and the rest of the team at Morgan Kaufmann who made this book happen. We love the art of Duane Bibby, whose cartoons enliven the chapters.

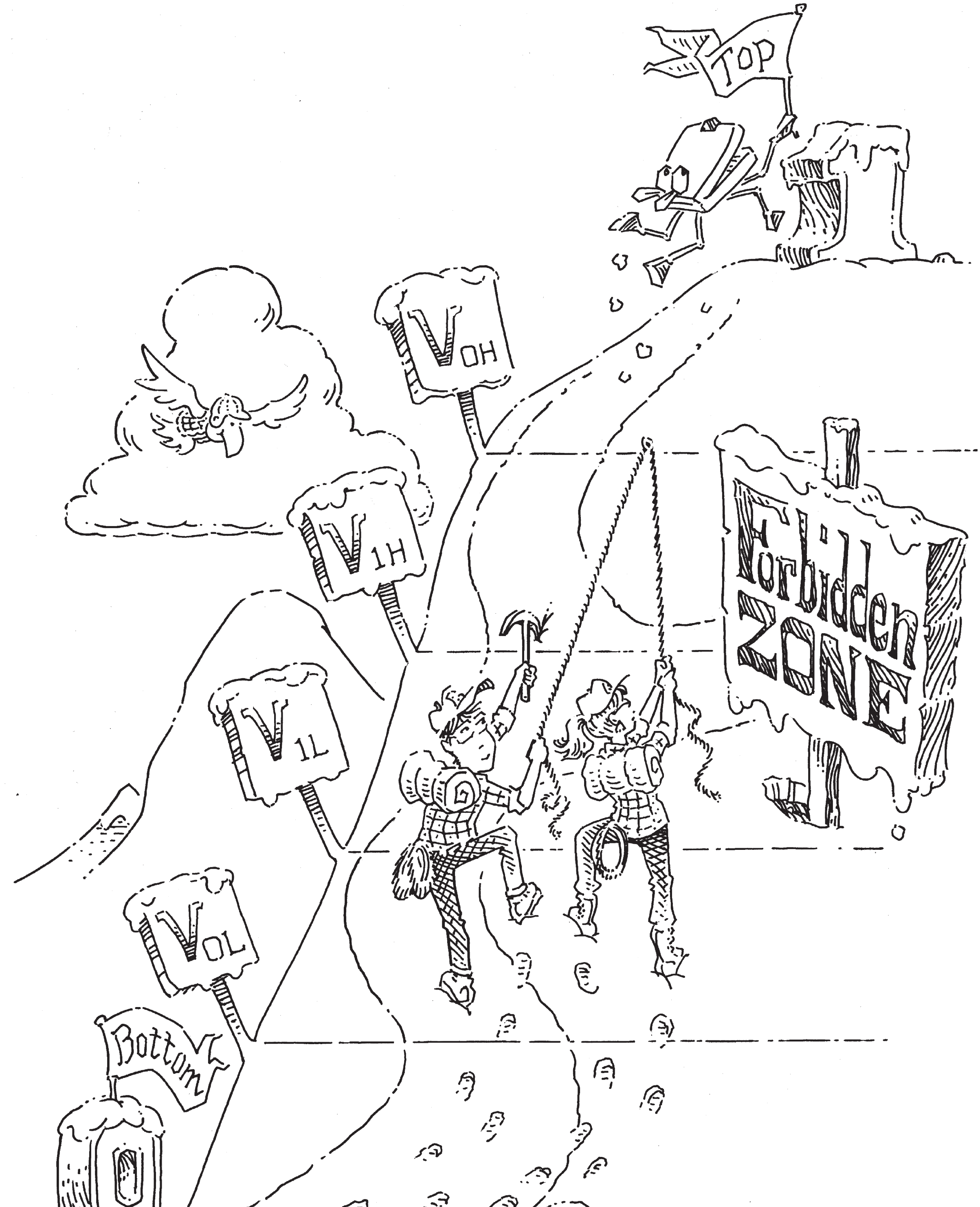
We thank Matthew Watkins, who contributed the section on Heterogeneous Multiprocessors in Chapter 7. We greatly appreciate the work of Joshua Vasquez, who developed code for the Raspberry Pi in Chapter 9. We also thank Josef Spjut and Ruye Wang, who class-tested the material.

Numerous reviewers substantially improved the book. They include Boyang Wang, John Barr, Jack V. Briner, Andrew C. Brown, Carl Baumgaertner, A. Utku Diril, Jim Frenzel, Jaeha Kim, Phillip King, James Pinter-Lucke, Amir Roth, Z. Jerry Shi, James E. Stine, Luke Teyssier, Peiyi Zhao, Zach Dodds, Nathaniel Guy, Aswin Krishna, Volnei Pedroni, Karl Wang, Ricardo Jasinski, Josef Spjut, Jörgen Lien, Sameer Sharma, John Nestor, Syed Manzoor, James Hoe, Srinivasa Vemuru, K. Joseph Hass, Jayantha Herath, Robert Mullins, Bruno Quoitin, Subramaniam Ganesan, Braden Phillips, John Oliver, Yahswant K. Malaiya, Mohammad Awedh, Zachary Kurmas, Donald Hung, and an anonymous reviewer. We appreciate Khaled Benkrid and his colleagues at ARM for their careful review of the ARM-related material.

We also appreciate the students in our courses at Harvey Mudd College and UNLV who have given us helpful feedback on drafts of this textbook. Of special note are Clinton Barnes, Matt Weiner, Carl Walsh, Andrew Carter, Casey Schilling, Alice Clifton, Chris Acon, and Stephen Brawner.

And last, but not least, we both thank our families for their love and support.

This page intentionally left blank

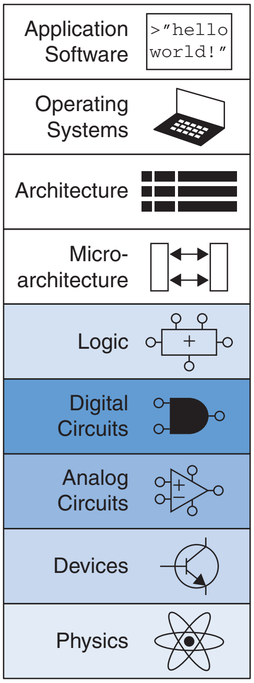


Top
V_{OH}
V_{1H}
Forbidden ZONE
V_{1L}
V_{OL}
Bottom
0

1

From Zero to One

1.1 The Game Plan
1.2 The Art of Managing Complexity
1.3 The Digital Abstraction
1.4 Number Systems
1.5 Logic Gates
1.6 Beneath the Digital Abstraction
1.7 CMOS Transistors*
1.8 Power Consumption*
1.9 Summary and a Look Ahead
Exercises
Interview Questions



1.1 THE GAME PLAN

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 100 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US $21 billion in 1985 to $306 billion in 2013, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and 0's as outputs. We then explore how to combine logic gates into more complicated modules such as adders and memories. Then we shift gears to programming in assembly language, the native tongue of the microprocessor. Finally, we put gates together to build a microprocessor that runs these assembly language programs.

A great advantage of digital systems is that the building blocks are quite simple: just 1's and 0's. They do not require grungy mathematics or a profound knowledge of physics. Instead, the designer's challenge is to combine these simple blocks into complicated systems. A microprocessor may be the first system that you build that is too complex to fit in

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00001-X
© 2013 Elsevier, Inc. All rights reserved.

your head all at once. One of the major themes weaved through this book is how to manage complexity.

1.2 THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to managing complexity. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

1.2.1 Abstraction

The critical technique for managing complexity is *abstraction:* hiding details when they are not important. A system can be viewed from many different levels of abstraction. For example, American politicians abstract the world into cities, counties, states, and countries. A county contains multiple cities and a state contains many counties. When a politician is running for president, the politician is mostly interested in how the state as a whole will vote, rather than how each county votes, so the state is the most useful level of abstraction. On the other hand, the Census Bureau measures the population of every city, so the agency must consider the details of a lower level of abstraction.

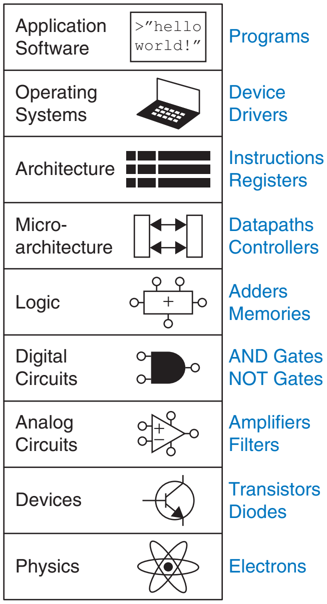

Application Software
>"hello world!"
Programs
Operating Systems
Device Drivers
Architecture
Instructions Registers
Micro-architecture
Datapaths Controllers
Logic
+
Adders Memories
Digital Circuits
AND Gates NOT Gates
Analog Circuits
+
−
Amplifiers Filters
Devices
Transistors Diodes
Physics
Electrons

Figure 1.1 **Levels of abstraction for an electronic computing system**

Figure 1.1 illustrates levels of abstraction for an electronic computer system along with typical building blocks at each level. At the lowest level of abstraction is the physics, the motion of electrons. The behavior of electrons is described by quantum mechanics and Maxwell's equations. Our system is constructed from electronic *devices* such as transistors (or vacuum tubes, once upon a time). These devices have well-defined connection points called *terminals* and can be modeled by the relationship between voltage and current as measured at each terminal. By abstracting to this device level, we can ignore the individual electrons. The next level of abstraction is *analog circuits*, in which devices are assembled to create components such as amplifiers. Analog circuits input and output a continuous range of voltages. *Digital circuits* such as logic gates restrict the voltages to discrete ranges, which we will use to indicate 0 and 1. In logic design, we build more complex structures, such as adders or memories, from digital circuits.

Microarchitecture links the logic and architecture levels of abstraction. The *architecture* level of abstraction describes a computer from the programmer's perspective. For example, the Intel x86 architecture used by microprocessors in most *personal computers* (*PCs*) is defined by a set of instructions and registers (memory for temporarily storing variables) that the programmer is allowed to use. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture. A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core i7, the Intel 80486, and the AMD Athlon all implement the x86 architecture with different microarchitectures.

Moving into the software realm, the operating system handles low-level details such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user. Thanks to the power of abstraction, your grandmother can surf the Web without any regard for the quantum vibrations of electrons or the organization of the memory in her computer.

This book focuses on the levels of abstraction from digital circuits through computer architecture. When you are working at one level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working. For example, a computer scientist cannot fully optimize code without understanding the architecture for which the program is being written. A device engineer cannot make wise trade-offs in transistor design without understanding the circuits in which the transistors will be used. We hope that by the time you finish reading this book, you can pick the level of abstraction appropriate to solving your problem and evaluate the impact of your design choices on other levels of abstraction.

Each chapter in this book begins with an abstraction icon indicating the focus of the chapter in deep blue, with secondary topics shown in lighter shades of blue.

1.2.2 Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline. One of the first examples of interchangeable parts was in flintlock rifle manufacturing. Until the early 19th century, rifles were individually crafted by hand. Components purchased from many different craftsmen were carefully filed and fit together by a highly skilled gunmaker. The discipline of interchangeable parts revolutionized the industry. By limiting the components to a standardized set with well-defined tolerances, rifles could be assembled and repaired much faster and with less skill. The gunmaker no longer concerned himself with lower levels of abstraction such as the specific shape of an individual barrel or gunstock.

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a subset of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design. By limiting ourselves to digital circuits, we can easily combine components into sophisticated systems that ultimately outperform those built from analog components in many applications. For example, digital televisions, compact disks (CDs), and cell phones are replacing their analog predecessors.

1.2.3 The Three-Y's

In addition to abstraction and discipline, designers use the three "-y's" to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

- *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- *Modularity* states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.
- *Regularity* seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

To illustrate these "-y's" we return to the example of rifle manufacturing. A flintlock rifle was one of the most intricate objects in common use in the early 19th century. Using the principle of hierarchy, we can break it into components shown in Figure 1.2: the lock, stock, and barrel.

The barrel is the long metal tube through which the bullet is fired. The lock is the firing mechanism. And the stock is the wooden body that holds the parts together and provides a secure grip for the user. In turn, the lock contains the trigger, hammer, flint, frizzen, and pan. Each of these components could be hierarchically described in further detail.

Modularity teaches that each component should have a well-defined function and interface. A function of the stock is to mount the barrel and lock. Its interface consists of its length and the location of its mounting pins. In a modular rifle design, stocks from many different manufacturers can be used with a particular barrel as long as the stock and barrel are of the correct length and have the proper mounting mechanism. A function of the barrel is to impart spin to the bullet so that it travels more accurately. Modularity dictates that there should be no side effects: the design of the stock should not impede the function of the barrel.

Regularity teaches that interchangeable parts are a good idea. With regularity, a damaged barrel can be replaced by an identical part. The barrels can be efficiently built on an assembly line, instead of being painstakingly hand-crafted.

We will return to these principles of hierarchy, modularity, and regularity throughout the book.

Captain Meriwether Lewis of the Lewis and Clark Expedition was one of the early advocates of interchangeable parts for rifles. In 1806, he explained:

> The guns of Drewyer and Sergt. Pryor were both out of order. The first was repared with a new lock, the old one having become unfit for use; the second had the cock screw broken which was replaced by a duplicate which had been prepared for the lock at Harpers Ferry where she was manufactured. But for the precaution taken in bringing on those extra locks, and parts of locks, in addition to the ingenuity of John Shields, most of our guns would at this moment be entirely unfit for use; but fortunately for us I have it in my power here to record that they are all in good order.

See Elliott Coues, ed., *The History of the Lewis and Clark Expedition…* (4 vols), New York: Harper, 1893; reprint, 3 vols, New York: Dover, 3:817.

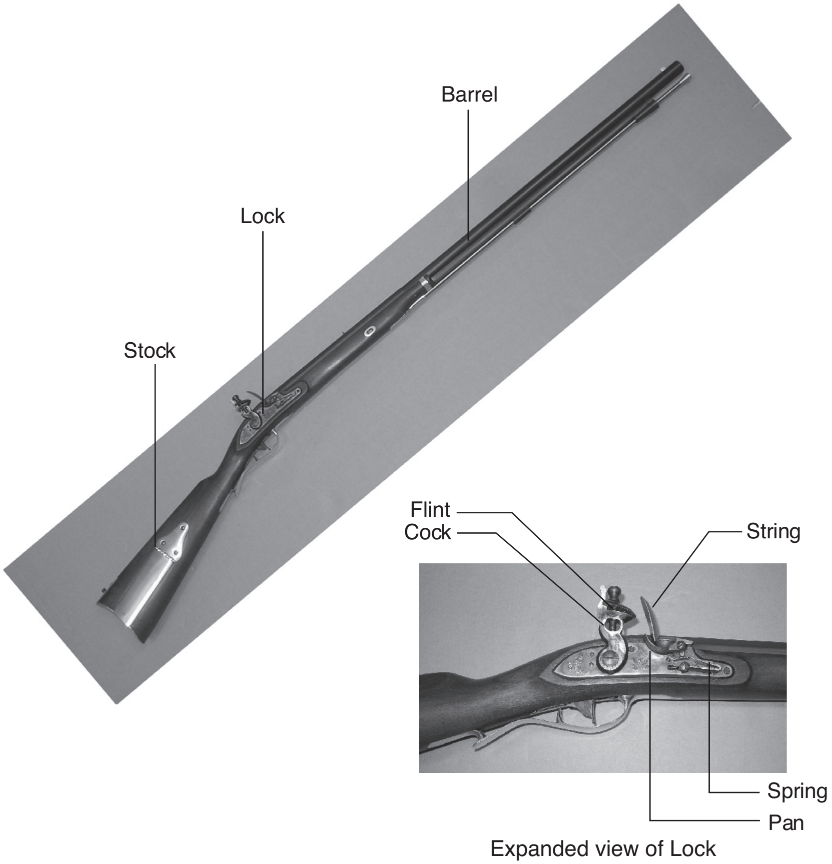


Figure 1.2 Flintlock rifle with a close-up view of the lock (Image by Euroarms Italia. *www.euroarms.net* © 2006.)

1.3 THE DIGITAL ABSTRACTION

Most physical variables are continuous. For example, the voltage on a wire, the frequency of an oscillation, or the position of a mass are all continuous quantities. Digital systems, on the other hand, represent information with *discrete-valued variables*—that is, variables with a finite number of distinct values.

An early digital system using variables with ten discrete values was Charles Babbage's Analytical Engine. Babbage labored from 1834 to 1871, designing and attempting to build this mechanical computer. The Analytical Engine used gears with ten positions labeled 0 through 9, much like a mechanical odometer in a car. Figure 1.3 shows a prototype of the Analytical Engine, in which each row processes one digit. Babbage chose 25 rows of gears, so the machine has 25-digit precision.

Charles Babbage, 1791–1871. Attended Cambridge University and married Georgiana Whitmore in 1814. Invented the Analytical Engine, the world's first mechanical computer. Also invented the cowcatcher and the universal postage rate. Interested in lock-picking, but abhorred street musicians (image courtesy of Fourmilab Switzerland, *www.fourmilab.ch*).

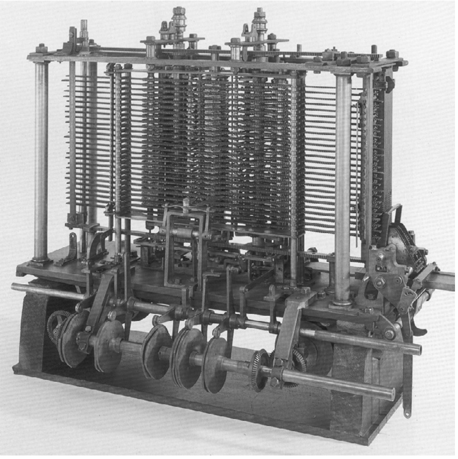


Figure 1.3 **Babbage's Analytical Engine, under construction at the time of his death in 1871** (image courtesy of Science Museum/Science and Society Picture Library)

George Boole, 1815–1864. Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote *An Investigation of the Laws of Thought* (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT (image courtesy of the American Institute of Physics).

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a '1' and a low voltage indicates a '0', because it is easier to distinguish between two voltages than ten.

The *amount of information* D in a discrete valued variable with N distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits} \tag{1.1}$$

A binary variable conveys $\log_2 2 = 1$ bit of information. Indeed, the word bit is short for *b*inary dig*it*. Each of Babbage's gears carried $\log_2 10 = 3.322$ bits of information because it could be in one of $2^{3.322} = 10$ unique positions. A continuous signal theoretically contains an infinite amount of information because it can take on an infinite number of values. In practice, noise and measurement error limit the information to only 10 to 16 bits for most continuous signals. If the measurement must be made rapidly, the information content is lower (e.g., 8 bits).

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent '1' and zero volts to represent '0'. In this book, we will use the terms '1', TRUE, and HIGH synonymously. Similarly, we will use '0', FALSE, and LOW interchangeably.

The beauty of the *digital abstraction* is that digital designers can focus on 1's and 0's, ignoring whether the Boolean variables are physically represented with specific voltages, rotating gears, or even hydraulic fluid levels. A computer programmer can work without needing to know the intimate details of the computer hardware. On the other hand, understanding the details of the hardware allows the programmer to optimize the software better for that specific computer.

An individual bit doesn't carry much information. In the next section, we examine how groups of bits can be used to represent numbers. In later chapters, we will also use groups of bits to represent letters and programs.

1.4 NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient. This section introduces the various number systems that will be used throughout the rest of the book.

1.4.1 Decimal Numbers

In elementary school, you learned to count and do arithmetic in *decimal*. Just as you (probably) have ten fingers, there are ten decimal digits: 0, 1, 2, …, 9. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are 1, 10, 100, 1000, and so on. Decimal numbers are referred to as *base 10*. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.4 shows how the decimal number 9742_{10} is written as the sum of each of its digits multiplied by the weight of the corresponding column.

An N-digit decimal number represents one of 10^N possibilities: 0, 1, 2, 3, …, $10^N - 1$. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

1.4.2 Binary Numbers

Bits represent one of two values, 0 or 1, and are joined together to form *binary numbers*. Each column of a binary number has twice the weight of the previous column, so binary numbers are *base 2*. In binary, the

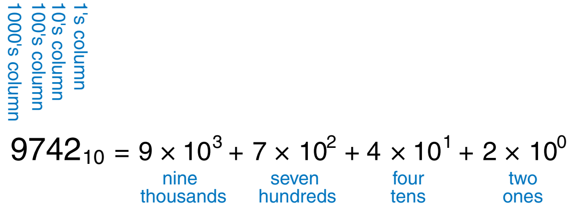


Figure 1.4 **Representation of a decimal number**

column weights (again from right to left) are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on. If you work with binary numbers often, you'll save time if you remember these powers of two up to 2^{16}.

An N-bit binary number represents one of 2^N possibilities: 0, 1, 2, 3, …, $2^N - 1$. Table 1.1 shows 1, 2, 3, and 4-bit binary numbers and their decimal equivalents.

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution: Figure 1.5 shows the conversion.

Table 1.1 **Binary numbers and their decimal equivalent**

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

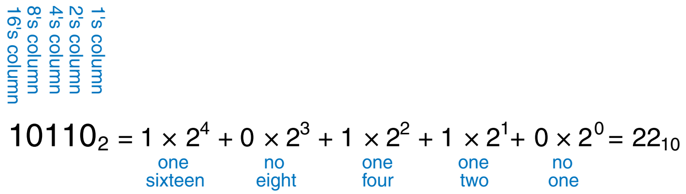

16's column
8's column
4's column
2's column
1's column
$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$
one sixteen
no eight
one four
one two
no one

Figure 1.5 **Conversion of a binary number to decimal**

Example 1.2 DECIMAL TO BINARY CONVERSION

Convert the decimal number 84_{10} to binary.

Solution: Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than or equal to the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving $84 - 64 = 20$. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving $20 - 16 = 4$. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving $4 - 4 = 0$. Thus there must be 0's in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. $84/2 = 42$, so 0 goes in the 1's column. $42/2 = 21$, so 0 goes in the 2's column. $21/2 = 10$ with a remainder of 1 going in the 4's column. $10/2 = 5$, so 0 goes in the 8's column. $5/2 = 2$ with a remainder of 1 going in the 16's column. $2/2 = 1$, so 0 goes in the 32's column. Finally $1/2 = 0$ with a remainder of 1 going in the 64's column. Again, $84_{10} = 1010100_2$.

1.4.3 Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of $2^4 = 16$ possibilities. Hence, it is sometimes more convenient to work in *base 16*, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F, as shown in Table 1.2. Columns in base 16 have weights of 1, 16, 16^2 (or 256), 16^3 (or 4096), and so on.

"Hexadecimal," a term coined by IBM in 1963, derives from the Greek *hexi* (six) and Latin *decem* (ten). A more proper term would use the Latin *sexa* (six), but *sexadecimal* sounded too risqué.

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

Convert the hexadecimal number $2ED_{16}$ to binary and to decimal.

Solution: Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. $2_{16} = 0010_2$, $E_{16} = 1110_2$ and $D_{16} = 1101_2$, so $2ED_{16} = 001011101101_2$. Conversion to decimal requires the arithmetic shown in Figure 1.6.

Table 1.2 Hexadecimal number system

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

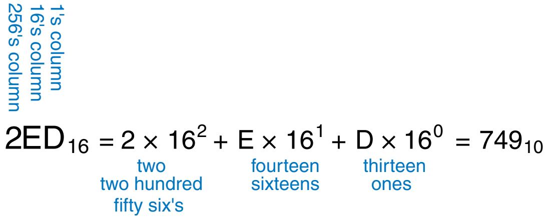

1's column
16's column
256's column
2ED16 = 2 × 16² + E × 16¹ + D × 16⁰ = 749₁₀
two two hundred fifty six's
fourteen sixteens
thirteen ones

Figure 1.6 Conversion of a hexadecimal number to decimal

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

Solution: Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence $1111010_2 = 7A_{16}$.

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution: Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than or equal to the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving $333 - 256 = 77$. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving $77 - 16 \times 4 = 13$. $13_{10} = \text{D}_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14\text{D}_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14\text{D}_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. $333/16 = 20$ with a remainder of $13_{10} = \text{D}_{16}$ going in the 1's column. $20/16 = 1$ with a remainder of 4 going in the 16's column. $1/16 = 0$ with a remainder of 1 going in the 256's column. Again, the result is 14D_{16}.

1.4.4 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a *byte*. It represents one of $2^8 = 256$ possibilities. The size of objects stored in computer memories is customarily measured in bytes rather than bits.

A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities. One hexadecimal digit stores one nibble and two hexadecimal digits store one full byte. Nibbles are no longer a commonly used unit, but the term is cute.

Microprocessors handle data in chunks called *words*. The size of a word depends on the architecture of the microprocessor. When this chapter was written in 2015, most computers had 64-bit processors, indicating that they operate on 64-bit words. At the time, older computers handling 32-bit words were also widely available. Simpler microprocessors, especially those used in gadgets such as toasters, use 8- or 16-bit words.

Within a group of bits, the bit in the 1's column is called the *least significant bit* (*lsb*), and the bit at the other end is called the *most significant bit* (*msb*), as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a word, the bytes are identified as *least significant byte* (*LSB*) through *most significant byte* (*MSB*), as shown in Figure 1.7(b) for a four-byte number written with eight hexadecimal digits.

A *microprocessor* is a processor built on a single chip. Until the 1970's, processors were too complicated to fit on one chip, so mainframe processors were built from boards containing many chips. Intel introduced the first 4-bit microprocessor, called the 4004, in 1971. Now, even the most sophisticated supercomputers are built using microprocessors. We will use the terms microprocessor and processor interchangeably throughout this book.

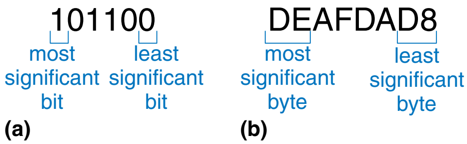


Figure 1.7 **Least and most significant bits and bytes**

By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 2^{10}. For example, 2^{10} bytes is one kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$. If you know $2^{10} \approx 1$ thousand, $2^{20} \approx 1$ million, $2^{30} \approx 1$ billion, and remember the powers of two up to 2^9, it is easy to estimate any power of two in your head.

Example 1.6 ESTIMATING POWERS OF TWO

Find the approximate value of 2^{24} without using a calculator.

Solution: Split the exponent into a multiple of ten and the remainder.

$2^{24} = 2^{20} \times 2^4$. $2^{20} \approx 1$ million. $2^4 = 16$. So $2^{24} \approx 16$ million. Technically, $2^{24} = 16{,}777{,}216$, but 16 million is close enough for marketing purposes.

1024 bytes is called a *kilobyte* (KB). 1024 bits is called a *kilobit* (Kb or Kbit). Similarly, MB, Mb, GB, and Gb are used for millions and billions of bytes and bits. Memory capacity is usually measured in bytes. Communication speed is usually measured in bits/sec. For example, the maximum speed of a dial-up modem is usually 56 kbits/sec.

1.4.5 Binary Addition

Binary addition is much like decimal addition, but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column. Figure 1.8 compares addition of decimal and binary numbers. In the right-most column of Figure 1.8(a), $7 + 9 = 16$, which cannot fit in a single digit because it is greater than 9. So we record the 1's digit, 6, and carry the 10's digit, 1, over to the next column. Likewise, in binary, if the sum of two numbers is greater than 1, we carry the 2's digit over to the next column. For example, in the right-most column of Figure 1.8(b),

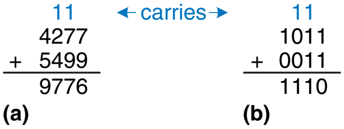


Figure 1.8 **Addition examples showing carries: (a) decimal (b) binary**

the sum $1 + 1 = 2_{10} = 10_2$ cannot fit in a single binary digit. So we record the 1's digit (0) and carry the 2's digit (1) of the result to the next column. In the second column, the sum is $1 + 1 + 1 = 3_{10} = 11_2$. Again, we record the 1's digit (1) and carry the 2's digit (1) to the next column. For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*.

Example 1.7 BINARY ADDITION

Compute $0111_2 + 0101_2$.

Solution: Figure 1.9 shows that the sum is 1100_2. The carries are indicated in blue. We can check our work by repeating the computation in decimal. $0111_2 = 7_{10}$. $0101_2 = 5_{10}$. The sum is $12_{10} = 1100_2$.

$$\begin{array}{r} 111 \\ 0111 \\ +\;0101 \\ \hline 1100 \end{array}$$

Figure 1.9 **Binary addition example**

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits. A 4-bit number, for example, has the range [0, 15]. 4-bit binary addition overflows if the result exceeds 15. The fifth bit is discarded, producing an incorrect result in the remaining four bits. Overflow can be detected by checking for a carry out of the most significant column.

Example 1.8 ADDITION WITH OVERFLOW

Compute $1101_2 + 0101_2$. Does overflow occur?

Solution: Figure 1.10 shows the sum is 10010_2. This result overflows the range of a 4-bit binary number. If it must be stored as four bits, the most significant bit is discarded, leaving the incorrect result of 0010_2. If the computation had been done using numbers with five or more bits, the result 10010_2 would have been correct.

$$\begin{array}{r} 111 \\ 1101 \\ +\;0101 \\ \hline 10010 \end{array}$$

Figure 1.10 **Binary addition example with overflow**

1.4.6 Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent *signed* binary numbers; the two most widely employed are called sign/magnitude and two's complement.

Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a minus sign followed by the magnitude. An N-bit sign/magnitude number uses the most significant bit as the sign and the remaining $N-1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

Example 1.9 SIGN/MAGNITUDE NUMBERS

Write 5 and −5 as 4-bit sign/magnitude numbers

Solution: Both numbers have a magnitude of $5_{10} = 101_2$. Thus, $5_{10} = 0101_2$ and $-5_{10} = 1101_2$.

Unfortunately, ordinary binary addition does not work for sign/magnitude numbers. For example, using ordinary addition on $-5_{10} + 5_{10}$ gives $1101_2 + 0101_2 = 10010_2$, which is nonsense.

An N-bit sign/magnitude number spans the range $[-2^{N-1} + 1, 2^{N-1} - 1]$. Sign/magnitude numbers are slightly odd in that both +0 and −0 exist. Both indicate zero. As you may expect, it can be troublesome to have two different representations for the same number.

The $7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

(Photograph courtesy of ESA/CNES/ARIANESPACE-Service Optique CS6.)

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00\ldots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\ldots111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\ldots000_2 = -2^{N-1}$. And −1 is written as all ones: $11\ldots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. This is useful to find the representation of a negative number or to determine the magnitude of a negative number.

Example 1.10 TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution: Start with $+2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

Find the decimal value of the two's complement number 1001_2.

Solution: 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N + 1^{\text{th}}$ result bit) is discarded.

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution: (a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

The two's complement of 0 is found by inverting all the bits (producing $11\ldots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate −0. Zero is considered positive because its sign bit is 0.

Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: −8 to 7. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no −0. The most negative number $10\ldots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\ldots111_2$) and adding 1, which produces $10\ldots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.

Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution: $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and −3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively.

Comparison of Number Systems

The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude. Table 1.3 compares the range of N-bit numbers in each of these three systems. Two's complement numbers are convenient because they represent both positive and negative integers and because ordinary addition works for all numbers. Subtraction is performed by negating the second number (i.e., taking the two's

Table 1.3 **Range of *N*-bit numbers**

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

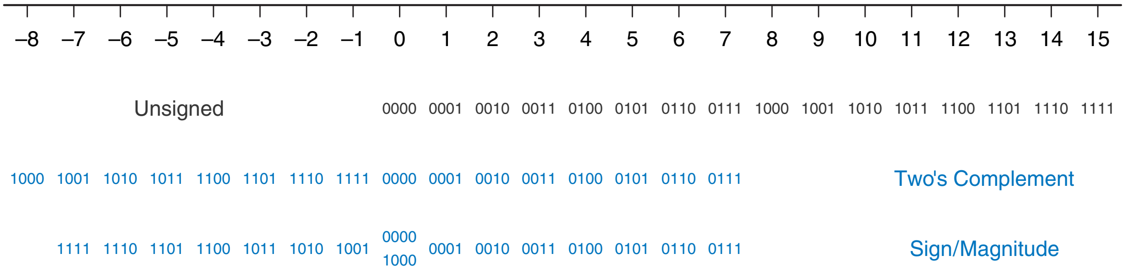

–8 –7 –6 –5 –4 –3 –2 –1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Unsigned
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111
Two's Complement
1111 1110 1101 1100 1011 1010 1001 0000 1000 0001 0010 0011 0100 0101 0110 0111
Sign/Magnitude

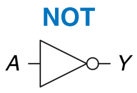
Figure 1.11 **Number line and 4-bit binary encodings**

complement), and then adding. Unless stated otherwise, assume that all signed binary numbers use two's complement representation.

Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range [0, 15] in regular binary order. Two's complement numbers span the range [−8, 7]. The nonnegative numbers [0, 7] share the same encodings as unsigned numbers. The negative numbers [−8, −1] are encoded such that a larger unsigned binary value represents a number closer to 0. Notice that the weird number, 1000, represents −8 and has no positive counterpart. Sign/magnitude numbers span the range [−7, 7]. The most significant bit is the sign bit. The positive numbers [1, 7] share the same encodings as unsigned numbers. The negative numbers are symmetric but have the sign bit set. 0 is represented by both 0000 and 1000. Thus, N-bit sign/magnitude numbers represent only $2^N - 1$ integers because of the two representations for 0.

1.5 LOGIC GATES

Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input (or inputs) and the output. Inputs are usually drawn on the left (or top) and outputs on the right (or bottom). Digital designers typically use letters near the beginning of the alphabet for gate inputs and the letter Y for the gate output. The relationship between the inputs and the output can be described with a truth table or a Boolean equation. A *truth table* lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A *Boolean equation* is a mathematical expression using binary variables.



$Y = \overline{A}$

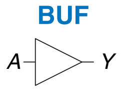
A	Y
0	1
1	0

Figure 1.12 **NOT gate**

1.5.1 NOT Gate

A *NOT gate* has one input, A, and one output, Y, as shown in Figure 1.12. The NOT gate's output is the inverse of its input. If A is FALSE, then Y is TRUE. If A is TRUE, then Y is FALSE. This relationship is summarized by the truth table and Boolean equation in the figure. The line over A in the Boolean equation is pronounced *NOT*, so $Y = \overline{A}$ is read "Y equals NOT A." The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$ or $Y = {\sim}A$. We will use $Y = \overline{A}$ exclusively, but don't be puzzled if you encounter another notation elsewhere.



$Y = A$

A	Y
0	0
1	1

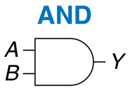
Figure 1.13 **Buffer**

1.5.2 Buffer

The other one-input logic gate is called a *buffer* and is shown in Figure 1.13. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. However, from the analog point of view, the buffer might have desirable characteristics such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates. This is an example of why we need to consider multiple levels of abstraction to fully understand a system; the digital abstraction hides the real purpose of a buffer.

The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1.14 **AND gate**

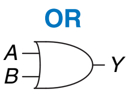
1.5.3 AND Gate

Two-input logic gates are more interesting. The *AND gate* shown in Figure 1.14 produces a TRUE output, Y, if and only if both A *and* B are TRUE. Otherwise, the output is FALSE. By convention, the inputs are listed in the order 00, 01, 10, 11, as if you were counting in binary. The Boolean equation for an AND gate can be written in several ways: $Y = A \bullet B$, $Y = AB$, or $Y = A \cap B$. The $\cap$ symbol is pronounced "intersection" and is preferred by logicians. We prefer $Y = AB$, read "Y equals A and B," because we are lazy.

According to Larry Wall, inventor of the Perl programming language, "the three principal virtues of a programmer are Laziness, Impatience, and Hubris."

1.5.4 OR Gate

The *OR gate* shown in Figure 1.15 produces a TRUE output, *Y*, if either *A* or *B* (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$. The ∪ symbol is pronounced union and is preferred by logicians. Digital designers normally use the + notation, $Y = A + B$ is pronounced "*Y* equals *A* or *B*."



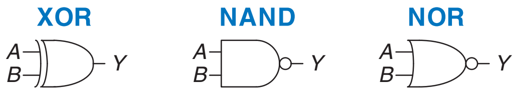
$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 1.15 OR gate

1.5.5 Other Two-Input Gates

Figure 1.16 shows other common two-input logic gates. *XOR* (exclusive OR, pronounced "ex-OR") is TRUE if *A* or *B*, but not both, are TRUE. The XOR operation is indicated by ⊕, a plus sign with a circle around it. Any gate can be followed by a bubble to invert its operation. The *NAND gate* performs NOT AND. Its output is TRUE unless both inputs are TRUE. The *NOR gate* performs NOT OR. Its output is TRUE if neither *A* nor *B* is TRUE. An *N*-input XOR gate is sometimes called a *parity* gate and produces a TRUE output if an odd number of inputs are TRUE. As with two-input gates, the input combinations in the truth table are listed in counting order.



$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = \overline{AB}$

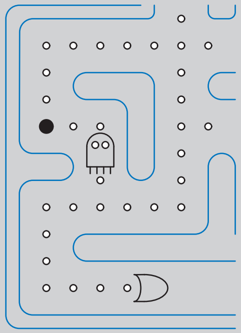
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 1.16 More two-input logic gates

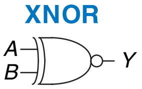
A silly way to remember the OR symbol is that its input side is curved like Pacman's mouth, so the gate is hungry and willing to eat any TRUE inputs it can find!



Example 1.15 XNOR GATE

Figure 1.17 shows the symbol and Boolean equation for a two-input *XNOR gate* that performs the inverse of an XOR. Complete the truth table.

Solution: Figure 1.18 shows the truth table. The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an *equality* gate because its output is TRUE when the inputs are equal.



$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

Figure 1.17 XNOR gate

1.5.6 Multiple-Input Gates

Many Boolean functions of three or more inputs exist. The most common are AND, OR, XOR, NAND, NOR, and XNOR. An *N*-input AND gate produces a TRUE output when all N inputs are TRUE. An N-input OR gate produces a TRUE output when at least one input is TRUE.

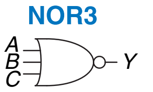
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.18 **XNOR truth table**

Example 1.16 THREE-INPUT NOR GATE

Figure 1.19 shows the symbol and Boolean equation for a three-input NOR gate. Complete the truth table.

Solution: Figure 1.20 shows the truth table. The output is TRUE only if none of the inputs are TRUE.



$Y = \overline{A + B + C}$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Figure 1.19 **Three-input NOR gate**

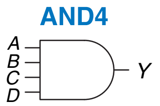
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 1.20 **Three-input NOR truth table**

Example 1.17 FOUR-INPUT AND GATE

Figure 1.21 shows the symbol and Boolean equation for a four-input AND gate. Create a truth table.

Solution: Figure 1.22 shows the truth table. The output is TRUE only if all of the inputs are TRUE.



$Y = ABCD$

Figure 1.21 **Four-input AND gate**

1.6 BENEATH THE DIGITAL ABSTRACTION

A digital system uses discrete-valued variables. However, the variables are represented by continuous physical quantities such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder. Hence, the designer must choose a way to relate the continuous value to the discrete value.

For example, consider representing a binary signal A with a voltage on a wire. Let 0 volts (V) indicate $A = 0$ and 5 V indicate $A = 1$. Any real system must tolerate some noise, so 4.97 V probably ought to be interpreted as $A = 1$ as well. But what about 4.3 V? Or 2.8 V? Or 2.500000 V?

1.6.1 Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called V_{DD}. In 1970's and 1980's technology, V_{DD} was generally 5 V. As chips have progressed to smaller transistors, V_{DD} has dropped to 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, or even lower to save power and avoid overloading the transistors.

1.6.2 Logic Levels

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown in Figure 1.23. The first gate is called the *driver* and the second gate is called the *receiver*. The output of the driver is connected to the input of the receiver. The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD}. If the receiver gets an input in the range of 0 to V_{IL}, it will consider the input to be LOW. If the receiver gets an input in the range of V_{IH} to V_{DD}, it will consider the input to be HIGH. If, for some reason such as noise or faulty components, the receiver's input should fall in the *forbidden zone* between V_{IL} and V_{IH}, the behavior of the gate is unpredictable. V_{OH}, V_{OL}, V_{IH}, and V_{IL} are called the output and input high and low logic levels.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 1.22 Four-input AND truth table

1.6.3 Noise Margins

If the output of the driver is to be correctly interpreted at the input of the receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$. Thus, even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level. The *noise margin* is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input. As can be seen in Figure 1.23, the low and high noise margins are, respectively

$$NM_L = V_{IL} - V_{OL} \tag{1.2}$$

$$NM_H = V_{OH} - V_{IH} \tag{1.3}$$

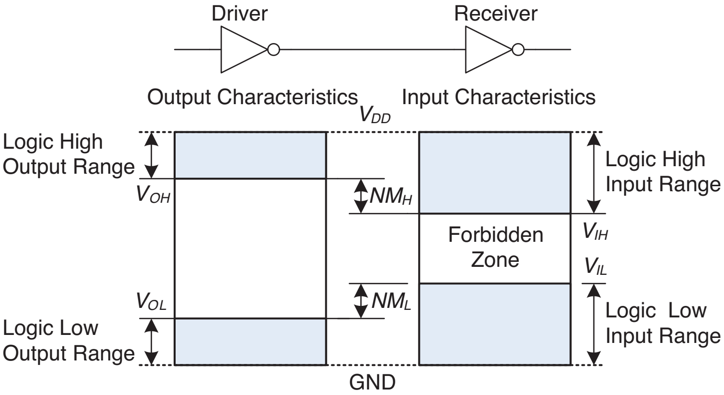


Figure 1.23 Logic levels and noise margins

Example 1.18 CALCULATING NOISE MARGINS

Consider the inverter circuit of Figure 1.24. V_{O1} is the output voltage of inverter I1, and V_{I2} is the input voltage of inverter I2. Both inverters have the following characteristics: $V_{DD} = 5$ V, $V_{IL} = 1.35$ V, $V_{IH} = 3.15$ V, $V_{OL} = 0.33$ V, and $V_{OH} = 3.84$ V. What are the inverter low and high noise margins? Can the circuit tolerate 1 V of noise between V_{O1} and V_{I2}?

V_{DD} stands for the voltage on the *drain* of a metal-oxide-semiconductor transistor, used to build most modern chips. The power supply voltage is also sometimes called V_{CC}, standing for the voltage on the *collector* of a bipolar junction transistor used to build chips in an older technology. Ground is sometimes called V_{SS} because it is the voltage on the *source* of a metal-oxide-semiconductor transistor. See Section 1.7 for more information on transistors.

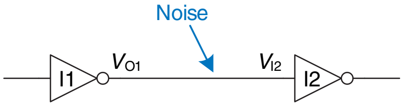


Figure 1.24 **Inverter circuit**

Solution: The inverter noise margins are: $NM_L = V_{IL} - V_{OL} = (1.35\text{ V} - 0.33\text{ V}) = 1.02\text{ V}$, $NM_H = V_{OH} - V_{IH} = (3.84\text{ V} - 3.15\text{ V}) = 0.69\text{ V}$. The circuit can tolerate 1 V of noise when the output is LOW ($NM_L = 1.02$ V) but not when the output is HIGH ($NM_H = 0.69$ V). For example, suppose the driver, I1, outputs its worst-case HIGH value, $V_{O1} = V_{OH} = 3.84$ V. If noise causes the voltage to droop by 1 V before reaching the input of the receiver, $V_{I2} = (3.84\text{ V} - 1\text{ V}) = 2.84$ V. This is less than the acceptable input HIGH value, $V_{IH} = 3.15$ V, so the receiver may not sense a proper HIGH input.

DC indicates behavior when an input voltage is held constant or changes slowly enough for the rest of the system to keep up. The term's historical root comes from *direct current*, a method of transmitting power across a line with a constant voltage. In contrast, the *transient response* of a circuit is the behavior when an input voltage changes rapidly. Section 2.9 explores transient response further.

1.6.4 DC Transfer Characteristics

To understand the limits of the digital abstraction, we must delve into the analog behavior of a gate. The DC *transfer characteristics* of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. They are called transfer characteristics because they describe the relationship between input and output voltages.

An ideal inverter would have an abrupt switching threshold at $V_{DD}/2$, as shown in Figure 1.25(a). For $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. For $V(A) > V_{DD}/2$, $V(Y) = 0$. In such a case, $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ and $V_{OL} = 0$.

A real inverter changes more gradually between the extremes, as shown in Figure 1.25(b). When the input voltage $V(A)$ is 0, the output voltage $V(Y) = V_{DD}$. When $V(A) = V_{DD}$, $V(Y) = 0$. However, the transition between these endpoints is smooth and may not be centered at exactly $V_{DD}/2$. This raises the question of how to define the logic levels.

A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y) / dV(A)$ is -1. These two points are called the *unity gain points*. Choosing logic levels at the unity gain points usually maximizes the noise margins. If V_{IL} were reduced, V_{OH} would only increase by a small amount. But if V_{IL} were increased, V_{OH} would drop precipitously.

1.6.5 The Static Discipline

To avoid inputs falling into the forbidden zone, digital logic gates are designed to conform to the *static discipline*. The static discipline requires that, given logically valid inputs, every circuit element will produce logically valid outputs.

By conforming to the static discipline, digital designers sacrifice the freedom of using arbitrary analog circuit elements in return for the simplicity and robustness of digital circuits. They raise the level of abstraction

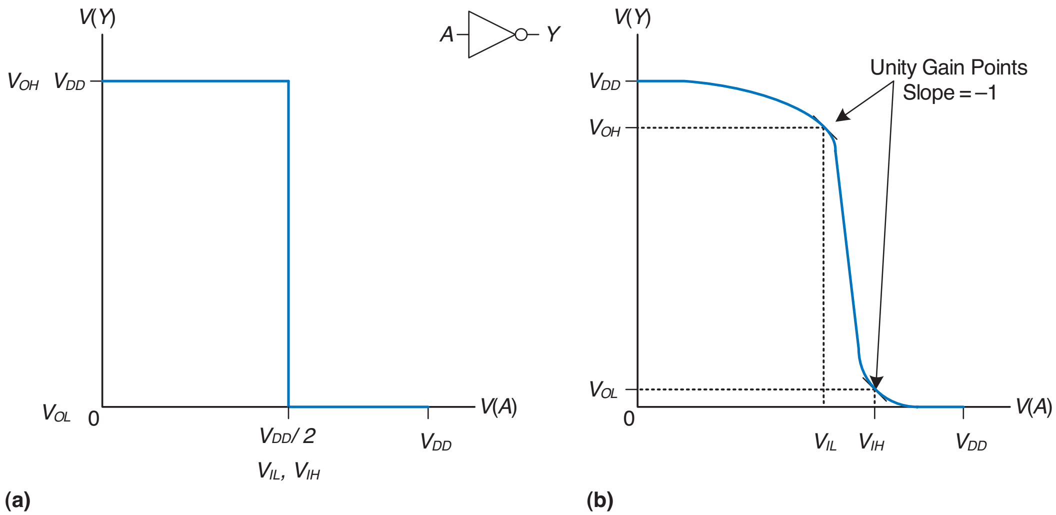


Figure 1.25 DC transfer characteristics and logic levels

from analog to digital, increasing design productivity by hiding needless detail.

The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Therefore, gates are grouped into *logic families* such that all gates in a logic family obey the static discipline when used with other gates in the family. Logic gates in the same logic family snap together like Legos in that they use consistent power supply voltages and logic levels.

Four major logic families that predominated from the 1970's through the 1990's are Transistor-Transistor Logic (TTL), Complementary Metal-Oxide-Semiconductor Logic (CMOS, pronounced sea-moss), Low Voltage TTL Logic (LVTTL), and Low Voltage CMOS Logic (LVCMOS). Their logic levels are compared in Table 1.4. Since then, logic families have balkanized with a proliferation of even lower power supply voltages. Appendix A.6 revisits popular logic families in more detail.

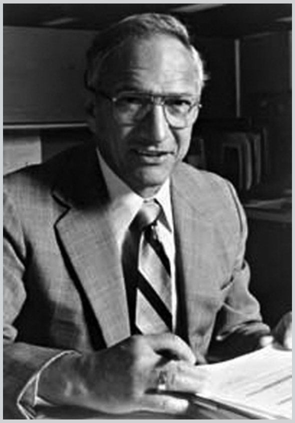
Table 1.4 Logic levels of 5 V and 3.3 V logic families

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Table 1.5 **Compatibility of logic families**

		Receiver TTL	 CMOS	 LVTTL	 LVCMOS
Driver	TTL	OK	NO: $V_{OH} < V_{IH}$	MAYBE[a]	MAYBE[a]
	CMOS	OK	OK	MAYBE[a]	MAYBE[a]
	LVTTL	OK	NO: $V_{OH} < V_{IH}$	OK	OK
	LVCMOS	OK	NO: $V_{OH} < V_{IH}$	OK	OK

[a] As long as a 5 V HIGH level does not damage the receiver input.



Robert Noyce, 1927–1990. Born in Burlington, Iowa. Received a B. A. in physics from Grinnell College and a Ph.D. in physics from MIT. Nicknamed "Mayor of Silicon Valley" for his profound influence on the industry.

Cofounded Fairchild Semiconductor in 1957 and Intel in 1968. Coinvented the integrated circuit. Many engineers from his teams went on to found other seminal semiconductor companies (photograph © 2006, Intel Corporation. Reproduced by permission).

Example 1.19 LOGIC FAMILY COMPATIBILITY

Which of the logic families in Table 1.4 can communicate with each other reliably?

Solution: Table 1.5 lists which logic families have compatible logic levels. Note that a 5 V logic family such as TTL or CMOS may produce an output voltage as HIGH as 5 V. If this 5 V signal drives the input of a 3.3 V logic family such as LVTTL or LVCMOS, it can damage the receiver, unless the receiver is specially designed to be "5-volt compatible."

1.7 CMOS TRANSISTORS*

This section and other sections marked with a * are optional and are not necessary to understand the main flow of the book.

Babbage's Analytical Engine was built from gears, and early electrical computers used relays or vacuum tubes. Modern computers use transistors because they are cheap, small, and reliable. *Transistors* are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. The two main types of transistors are *bipolar junction transistors* and *metal-oxide-semiconductor field effect transistors* (*MOSFETs* or *MOS transistors*, pronounced "moss-fets" or "M-O-S", respectively).

In 1958, Jack Kilby at Texas Instruments built the first integrated circuit containing two transistors. In 1959, Robert Noyce at Fairchild Semiconductor patented a method of interconnecting multiple transistors on a single silicon chip. At the time, transistors cost about $10 each.

Thanks to more than four decades of unprecedented manufacturing advances, engineers can now pack roughly three billion MOSFETs onto a 1 cm^2 chip of silicon, and these transistors cost less than 1 microcent apiece. The capacity and cost continue to improve by an order of magnitude every 8 years or so. MOSFETs are now the building blocks of almost all digital systems. In this section, we will peer beneath the digital abstraction to see how logic gates are built from MOSFETs.

1.7.1 Semiconductors

MOS transistors are built from silicon, the predominant atom in rock and sand. Silicon (Si) is a group IV atom, so it has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline *lattice*. Figure 1.26(a) shows the lattice in two dimensions for ease of drawing, but remember that the lattice actually forms a cubic crystal. In the figure, a line represents a covalent bond. By itself, silicon is a poor conductor because all the electrons are tied up in covalent bonds. However, it becomes a better conductor when small amounts of impurities, called *dopant* atoms, are carefully added. If a group V dopant such as arsenic (As) is added, the dopant atoms have an extra electron that is not involved in the bonds. The electron can easily move about the lattice, leaving an ionized dopant atom (As^+) behind, as shown in Figure 1.26(b). The electron carries a negative charge, so we call arsenic an *n-type* dopant. On the other hand, if a group III dopant such as boron (B) is added, the dopant atoms are missing an electron, as shown in Figure 1.26(c). This missing electron is called a *hole*. An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B^-) and leaving a hole at the neighboring silicon atom. In a similar fashion, the hole can migrate around the lattice. The hole is a lack of negative charge, so it acts like a positively charged particle. Hence, we call boron a *p-type* dopant. Because the conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants, silicon is called a *semiconductor*.

1.7.2 Diodes

The junction between p-type and n-type silicon is called a *diode*. The p-type region is called the *anode* and the n-type region is called the *cathode*, as illustrated in Figure 1.27. When the voltage on the anode rises above the voltage on the cathode, the diode is *forward biased*, and current

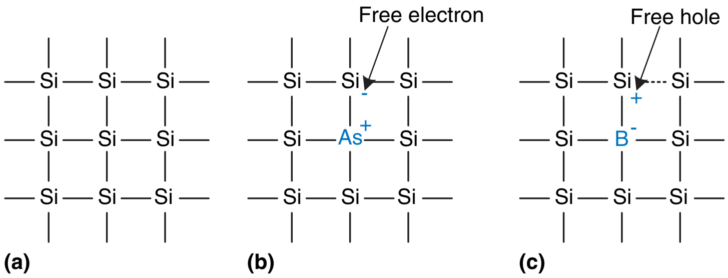


Figure 1.26 Silicon lattice and dopant atoms

flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is *reverse biased*, and no current flows. The diode symbol intuitively shows that current only flows in one direction.

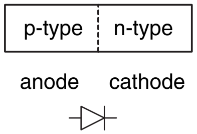


Figure 1.27 **The p-n junction diode structure and symbol**

1.7.3 Capacitors

A *capacitor* consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$. The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = Q/V$. The capacitance is proportional to the size of the conductors and inversely proportional to the distance between them. The symbol for a capacitor is shown in Figure 1.28.

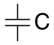
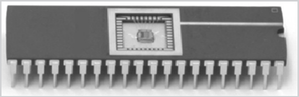


Figure 1.28 **Capacitor symbol**

Capacitance is important because charging or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate. Speed and energy will be discussed throughout this book.

Technicians in an Intel clean room wear Gore-Tex bunny suits to prevent particulates from their hair, skin, and clothing from contaminating the microscopic transistors on silicon wafers (photograph © 2006, Intel Corporation. Reproduced by permission).



A 40-pin dual-inline package (DIP) contains a small chip (scarcely visible) in the center that is connected to 40 metal pins, 20 on a side, by gold wires thinner than a strand of hair (photograph by Kevin Mapp. © 2006 Harvey Mudd College).

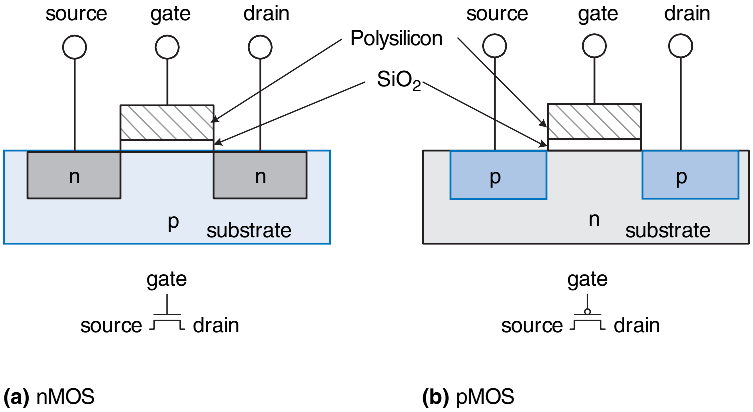
1.7.4 nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron[1] in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called *chips* or *dice* that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.

The MOSFET sandwich consists of a conducting layer called the *gate* on top of an insulating layer of *silicon dioxide* (SiO_2) on top of the silicon wafer, called the *substrate*. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called *oxide* in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a *dielectric* separates the metal and semiconductor plates.

---

[1] $1\ \mu m = 1$ micron $= 10^{-6}$ m.



(a) nMOS **(b)** pMOS

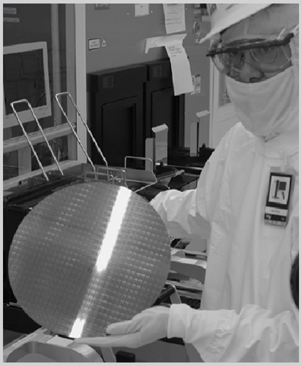
Figure 1.29 **nMOS and pMOS transistors**

The source and drain terminals are physically symmetric. However, we say that charge flows from the source to the drain. In an nMOS transistor, the charge is carried by electrons, which flow from negative voltage to positive voltage. In a pMOS transistor, the charge is carried by holes, which flow from positive voltage to negative voltage. If we draw schematics with the most positive voltage at the top and the most negative at the bottom, the source of (negative) charges in an nMOS transistor is the bottom terminal and the source of (positive) charges in a pMOS transistor is the top terminal.

There are two flavors of MOSFETs: nMOS and pMOS (pronounced "n-moss" and "p-moss"). Figure 1.29 shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent to the gate called the *source* and the *drain* and are built on a p-type semiconductor substrate. The *pMOS* transistors are just the opposite, consisting of p-type source and drain regions in an n-type *substrate*.

A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, the lowest voltage in the system. First, consider the situation when the gate is also at 0 V, as shown in Figure 1.30(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in Figure 1.30(b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the *channel*. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the *threshold voltage*, V_t, and is typically 0.3 to 0.7 V.



A technician holds a 12-inch wafer containing hundreds of microprocessor chips (photograph © 2006, Intel Corporation. Reproduced by permission).

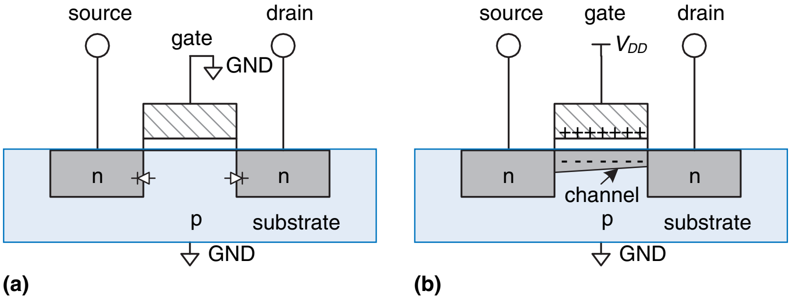
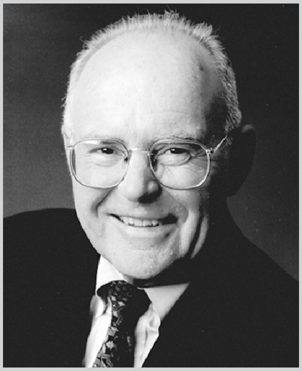


Figure 1.30 **nMOS transistor operation**



Gordon Moore, 1929–. Born in San Francisco. Received a B.S. in chemistry from UC Berkeley and a Ph.D. in chemistry and physics from Caltech. Cofounded Intel in 1968 with Robert Noyce. Observed in 1965 that the number of transistors on a computer chip doubles every year. This trend has become known as *Moore's Law*. Since 1975, transistor counts have doubled every two years.

A corollary of Moore's Law is that microprocessor performance doubles every 18 to 24 months. Semiconductor sales have also increased exponentially.

Moore's Law has driven the incredible advances of the semiconductor industry for 50 years as the feature size of transistors has dropped from more then 10 μm to only 28 nm. However, this progress is showing signs of slowing below the 28 nm node because building transistors much smaller than the wavelength of light is expensive. (Photograph © 2006, Intel Corporation. Reproduced by permission.)

pMOS transistors work in just the opposite fashion, as might be guessed from the bubble on their symbol shown in Figure 1.31. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the drain will only swing between 0 and $V_{DD} - V_t$. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called *wells* where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or CMOS. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS processes give us two types of electrically controlled switches, as shown in Figure 1.31. The voltage at the gate (*g*) regulates the flow of current between the source (*s*) and drain (*d*). nMOS

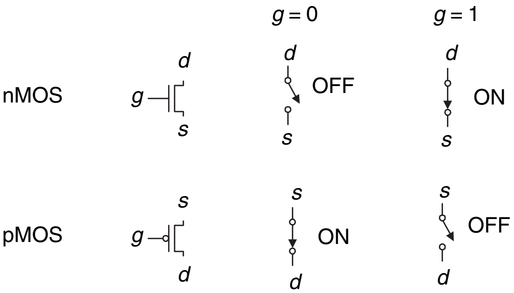


Figure 1.31 **Switch models of MOSFETs**

transistors are OFF when the gate is 0 and ON when the gate is 1. pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

1.7.5 CMOS NOT Gate

Figure 1.32 shows a schematic of a NOT gate built with CMOS transistors. The triangle indicates GND, and the flat bar indicates V_{DD}; these labels will be omitted from future schematics. The nMOS transistor, N1, is connected between GND and the *Y* output. The pMOS transistor, P1, is connected between V_{DD} and the *Y* output. Both transistor gates are controlled by the input, *A*.

If $A = 0$, N1 is OFF and P1 is ON. Hence, *Y* is connected to V_{DD} but not to GND, and is pulled up to a logic 1. P1 passes a good 1. If $A = 1$, N1 is ON and P1 is OFF, and *Y* is pulled down to a logic 0. N1 passes a good 0. Checking against the truth table in Figure 1.12, we see that the circuit is indeed a NOT gate.

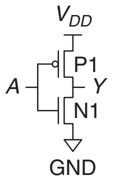


Figure 1.32 **NOT gate schematic**

1.7.6 Other CMOS Logic Gates

Figure 1.33 shows a schematic of a two-input NAND gate. In schematic diagrams, wires are always joined at three-way junctions. They are joined at four-way junctions only if a dot is shown. The nMOS transistors N1 and N2 are connected in series; both nMOS transistors must be ON to pull the output down to GND. The pMOS transistors P1 and P2 are in parallel; only one pMOS transistor must be ON to pull the output up to V_{DD}. Table 1.6 lists the operation of the pull-down and pull-up networks and the state of the output, demonstrating that the gate does function as a NAND. For example, when $A = 1$ and $B = 0$, N1 is ON, but N2 is OFF, blocking the path from *Y* to GND. P1 is OFF, but P2 is ON, creating a path from V_{DD} to *Y*. Therefore, *Y* is pulled up to 1.

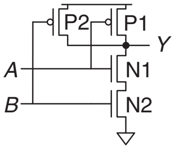


Figure 1.33 **Two-input NAND gate schematic**

Figure 1.34 shows the general form used to construct any inverting logic gate, such as NOT, NAND, or NOR. nMOS transistors are good at passing 0's, so a pull-down network of nMOS transistors is placed between the output and GND to pull the output down to 0. pMOS transistors are

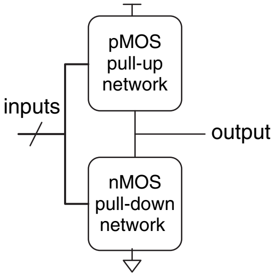


Figure 1.34 **General form of an inverting logic gate**

Table 1.6 **NAND gate operation**

A	*B*	Pull-Down Network	Pull-Up Network	*Y*
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

Experienced designers claim that electronic devices operate because they contain *magic smoke*. They confirm this theory with the observation that if the magic smoke is ever let out of the device, it ceases to work.

good at passing 1's, so a pull-up network of pMOS transistors is placed between the output and V_{DD} to pull the output up to 1. The networks may consist of transistors in series or in parallel. When transistors are in parallel, the network is ON if either transistor is ON. When transistors are in series, the network is ON only if both transistors are ON. The slash across the input wire indicates that the gate may receive multiple inputs.

If both the pull-up and pull-down networks were ON simultaneously, a *short circuit* would exist between V_{DD} and GND. The output of the gate might be in the forbidden zone and the transistors would consume large amounts of power, possibly enough to burn out. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output *floats*. Its value is again undefined. Floating outputs are usually undesirable, but in Section 2.6 we will see how they can occasionally be used to the designer's advantage.

In a properly functioning logic gate, one of the networks should be ON and the other OFF at any given time, so that the output is pulled HIGH or LOW but not shorted or floating. We can guarantee this by using the rule of *conduction complements*. When nMOS transistors are in series, the pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

Example 1.20 THREE-INPUT NAND SCHEMATIC

Draw a schematic for a three-input NAND gate using CMOS transistors.

Solution: The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel. Such a gate is shown in Figure 1.35; you can verify the function by checking that it has the correct truth table.

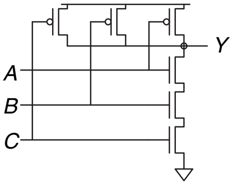


Figure 1.35 **Three-input NAND gate schematic**

Example 1.21 TWO-INPUT NOR SCHEMATIC

Draw a schematic for a two-input NOR gate using CMOS transistors.

Solution: The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series. Such a gate is shown in Figure 1.36.

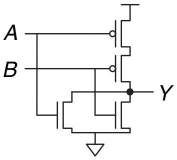


Figure 1.36 **Two-input NOR gate schematic**

Example 1.22 TWO-INPUT AND SCHEMATIC

Draw a schematic for a two-input AND gate.

Solution: It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT, as shown in Figure 1.37.

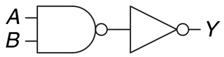


Figure 1.37 Two-input AND gate schematic

1.7.7 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well. Recall that nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well. Figure 1.38 shows such a circuit, called a *transmission gate* or *pass gate*. The two sides of the switch are called A and B because a switch is bidirectional and has no preferred input or output side. The control signals are called *enables*, EN and $\overline{EN}$. When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so A and B are not connected. When $EN = 1$ and $\overline{EN} = 0$, the transmission gate is ON or enabled, and any logic value can flow between A and B.

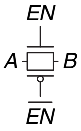


Figure 1.38 Transmission gate

1.7.8 Pseudo-nMOS Logic

An N-input CMOS NOR gate uses N nMOS transistors in parallel and N pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel. Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore the parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.39. This pMOS transistor is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output Y HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls Y down close enough to GND to produce a logic 0.

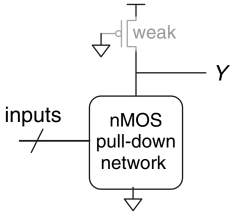


Figure 1.39 Generic pseudo-nMOS gate

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs. For example, Figure 1.40 shows a pseudo-nMOS four-input NOR. Pseudo-nMOS gates are useful for certain memory and logic arrays discussed in Chapter 5. The disadvantage is that a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

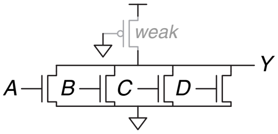


Figure 1.40 Pseudo-nMOS four-input NOR gate

Pseudo-nMOS gates got their name from the 1970's, when manufacturing processes only had nMOS transistors. A weak nMOS transistor was used to pull the output HIGH because pMOS transistors were not available.

1.8 POWER CONSUMPTION*

Power consumption is the amount of energy used per unit time. Power consumption is of great importance in digital systems. The battery life of portable systems such as cell phones and laptop computers is limited by power consumption. Power is also significant for systems that are plugged in, because electricity costs money and because the system will overheat if it draws too much power.

Digital systems draw both *dynamic* and *static* power. Dynamic power is the power used to charge capacitance as signals change between 0 and 1. Static power is the power used even when signals do not change and the system is idle.

Logic gates and the wires that connect them have capacitance. The energy drawn from the power supply to charge a capacitance C to voltage V_{DD} is CV_{DD}^2. If the voltage on the capacitor switches at frequency f (i.e., f times per second), it charges the capacitor $f/2$ times and discharges it $f/2$ times per second. Discharging does not draw energy from the power supply, so the dynamic power consumption is

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^2 f \tag{1.4}$$

Electrical systems draw some current even when they are idle. When transistors are OFF, they leak a small amount of current. Some circuits, such as the pseudo-nMOS gate discussed in Section 1.7.8, have a path from V_{DD} to GND through which current flows continuously. The total static current, I_{DD}, is also called the *leakage current* or the *quiescent supply current* flowing between V_{DD} and GND. The static power consumption is proportional to this static current:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1.5}$$

Example 1.23 POWER CONSUMPTION

A particular cell phone has a 6 watt-hour (W-hr) battery and operates at 1.2 V. Suppose that, when it is in use, the cell phone operates at 300 MHz and the average amount of capacitance in the chip switching at any given time is 10 nF (10^{-8} Farads). When in use, it also broadcasts 3 W of power out of its antenna. When the phone is not in use, the dynamic power drops to almost zero because the signal processing is turned off. But the phone also draws 40 mA of quiescent current whether it is in use or not. Determine the battery life of the phone (a) if it is not being used, and (b) if it is being used continuously.

Solution: The static power is $P_{\text{static}} = (0.040\ \text{A})(1.2\ \text{V}) = 48\ \text{mW}$. (a) If the phone is not being used, this is the only power consumption, so the battery life is (6 Whr)/(0.048 W) = 125 hours (about 5 days). (b) If the phone is being used, the dynamic power is $P_{\text{dynamic}} = (0.5)(10^{-8}\ \text{F})(1.2\ \text{V})^2(3 \times 10^8\ \text{Hz}) = 2.16\ \text{W}$. Together with the static and broadcast power, the total active power is $2.16\ \text{W} + 0.048\ \text{W} + 3\ \text{W} = 5.2\ \text{W}$, so the battery life is $6\ \text{W-hr}/5.2\ \text{W} = 1.15$ hours. This example somewhat oversimplifies the actual operation of a cell phone, but it illustrates the key ideas of power consumption.

1.9 SUMMARY AND A LOOK AHEAD

There are 10 kinds of people in this world: those who can count in binary and those who can't.

This chapter has introduced principles for understanding and designing complex systems. Although the real world is analog, digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH. Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

- **NOT:** TRUE when input is FALSE
- **AND:** TRUE when all inputs are TRUE
- **OR:** TRUE when any inputs are TRUE
- **XOR:** TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches. nMOS transistors turn ON when the gate is 1. pMOS transistors turn ON when the gate is 0.

In Chapters 2 through 5, we continue the study of digital logic. Chapter 2 addresses *combinational logic*, in which the outputs depend only on the current inputs. The logic gates introduced already are examples of combinational logic. You will learn to design circuits involving multiple gates to implement a relationship between inputs and outputs specified by a truth table or Boolean equation. Chapter 3 addresses *sequential logic*, in which the outputs depend on both current and past inputs. *Registers* are common sequential elements that remember their previous input. *Finite state machines*, built from registers and combinational logic, are a powerful way to build complicated systems in a systematic fashion. We also study timing of digital systems to analyze how fast a system can operate. Chapter 4 describes hardware description languages (HDLs). HDLs are related to conventional programming languages but are used to simulate and build hardware rather than software. Most digital systems today are designed with HDLs. SystemVerilog and VHDL are the two prevalent languages, and they are covered side-by-side in this book. Chapter 5 studies other combinational and sequential building blocks such as adders, multipliers, and memories.

Chapter 6 shifts to computer architecture. It describes the ARM processor, an industry-standard microprocessor used in almost all smart phones and tablets and many other devices, from pinball machines to cars and servers. The ARM architecture is defined by its registers and assembly language instruction set. You will learn to write programs in assembly language for the ARM processor so that you can communicate with the processor in its native language.

Chapters 7 and 8 bridge the gap between digital logic and computer architecture. Chapter 7 investigates microarchitecture, the arrangement of digital building blocks, such as adders and registers, needed to construct a processor. In that chapter, you learn to build your own ARM processor. Indeed, you learn three microarchitectures illustrating different trade-offs of performance and cost. Processor performance has increased exponentially, requiring ever more sophisticated memory systems to feed the insatiable demand for data. Chapter 8 delves into memory system architecture. Chapter 9 (available as a web supplement, see Preface) describes how computers communicate with peripheral devices such as monitors, Bluetooth radios, and motors.

Exercises

Exercise 1.1 Explain in one paragraph at least three levels of abstraction that are used by

(a) biologists studying the operation of cells.

(b) chemists studying the composition of matter.

Exercise 1.2 Explain in one paragraph how the techniques of hierarchy, modularity, and regularity may be used by

(a) automobile designers.

(b) businesses to manage their operations.

Exercise 1.3 Ben Bitdiddle is building a house. Explain how he can use the principles of hierarchy, modularity, and regularity to save time and money during construction.

Exercise 1.4 An analog voltage is in the range of 0–5 V. If it can be measured with an accuracy of ±50 mV, at most how many bits of information does it convey?

Exercise 1.5 A classroom has an old clock on the wall whose minute hand broke off.

(a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time?

(b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

Exercise 1.6 The Babylonians developed the *sexagesimal* (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the number 4000_{10} in sexagesimal?

Exercise 1.7 How many different numbers can be represented with 16 bits?

Exercise 1.8 What is the largest unsigned 32-bit binary number?

Exercise 1.9 What is the largest 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.10 What is the largest 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.11 What is the smallest (most negative) 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.12 What is the smallest (most negative) 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.13 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1010_2

(b) 110110_2

(c) 11110000_2

(d) 0001000101001111_2

Exercise 1.14 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1110_2

(b) 100100_2

(c) 11010111_2

(d) 0111010101001000_2

Exercise 1.15 Repeat Exercise 1.13, but convert to hexadecimal.

Exercise 1.16 Repeat Exercise 1.14, but convert to hexadecimal.

Exercise 1.17 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $A5_{16}$

(b) $3B_{16}$

(c) $FFFF_{16}$

(d) $D0000000_{16}$

Exercise 1.18 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $4E_{16}$

(b) $7C_{16}$

(c) $ED3A_{16}$

(d) $403FB001_{16}$

Exercise 1.19 Repeat Exercise 1.17, but convert to unsigned binary.

Exercise 1.20 Repeat Exercise 1.18, but convert to unsigned binary.

Exercise 1.21 Convert the following two's complement binary numbers to decimal.

(a) 1010_2

(b) 110110_2

(c) 01110000_2

(d) 10011111_2

Exercise 1.22 Convert the following two's complement binary numbers to decimal.

(a) 1110_2

(b) 100011_2

(c) 01001110_2

(d) 10110101_2

Exercise 1.23 Repeat Exercise 1.21, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.24 Repeat Exercise 1.22, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.25 Convert the following decimal numbers to unsigned binary numbers.

(a) 42_{10}

(b) 63_{10}

(c) 229_{10}

(d) 845_{10}

Exercise 1.26 Convert the following decimal numbers to unsigned binary numbers.

(a) 14_{10}

(b) 52_{10}

(c) 339_{10}

(d) 711_{10}

Exercise 1.27 Repeat Exercise 1.25, but convert to hexadecimal.

Exercise 1.28 Repeat Exercise 1.26, but convert to hexadecimal.

Exercise 1.29 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 42_{10}

(b) -63_{10}

(c) 124_{10}

(d) -128_{10}

(e) 133_{10}

Exercise 1.30 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 24_{10}

(b) -59_{10}

(c) 128_{10}

(d) -150_{10}

(e) 127_{10}

Exercise 1.31 Repeat Exercise 1.29, but convert to 8-bit sign/magnitude numbers.

Exercise 1.32 Repeat Exercise 1.30, but convert to 8-bit sign/magnitude numbers.

Exercise 1.33 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0101_2

(b) 1010_2

Exercise 1.34 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0111_2

(b) 1001_2

Exercise 1.35 Repeat Exercise 1.33 if the numbers are unsigned rather than two's complement.

Exercise 1.36 Repeat Exercise 1.34 if the numbers are unsigned rather than two's complement.

Exercise 1.37 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.25 to octal.

Exercise 1.38 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.26 to octal.

Exercise 1.39 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 42_8

(b) 63_8

(c) 255_8

(d) 3047_8

Exercise 1.40 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 23_8

(b) 45_8

(c) 371_8

(d) 2560_8

Exercise 1.41 How many 5-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.42 How many 7-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.43 How many bytes are in a 32-bit word? How many nibbles are in the word?

Exercise 1.44 How many bytes are in a 64-bit word?

Exercise 1.45 A particular DSL modem operates at 768 kbits/sec. How many bytes can it receive in 1 minute?

Exercise 1.46 USB 3.0 can send data at 5 Gbits/sec. How many bytes can it send in 1 minute?

Exercise 1.47 Hard disk manufacturers use the term "megabyte" to mean 10^6 bytes and "gigabyte" to mean 10^9 bytes. How many real GBs of music can you store on a 50 GB hard disk?

Exercise 1.48 Estimate the value of 2^{31} without using a calculator.

Exercise 1.49 A memory on the Pentium II microprocessor is organized as a rectangular array of bits with 2^8 rows and 2^9 columns. Estimate how many bits it has without using a calculator.

Exercise 1.50 Draw a number line analogous to Figure 1.11 for 3-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.51 Draw a number line analogous to Figure 1.11 for 2-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.52 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows a 4-bit result.

(a) $1001_2 + 0100_2$

(b) $1101_2 + 1011^2$

Exercise 1.53 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows an 8-bit result.

(a) $10011001_2 + 01000100_2$

(b) $11010010_2 + 10110110_2$

Exercise 1.54 Repeat Exercise 1.52, assuming that the binary numbers are in two's complement form.

Exercise 1.55 Repeat Exercise 1.53, assuming that the binary numbers are in two's complement form.

Exercise 1.56 Convert the following decimal numbers to 6-bit two's complement binary numbers and add them. Indicate whether or not the sum overflows a 6-bit result.

(a) $16_{10} + 9_{10}$

(b) $27_{10} + 31_{10}$

(c) $-4_{10} + 19_{10}$

(d) $3_{10} + -32_{10}$

(e) $-16_{10} + -9_{10}$

(f) $-27_{10} + -31_{10}$

Exercise 1.57 Repeat Exercise 1.56 for the following numbers.

(a) $7_{10} + 13_{10}$

(b) $17_{10} + 25_{10}$

(c) $-26_{10} + 8_{10}$

(d) $31_{10} + -14_{10}$

(e) $-19_{10} + -22_{10}$

(f) $-2_{10} + -29_{10}$

Exercise 1.58 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

(a) $7_{16} + 9_{16}$

(b) $13_{16} + 28_{16}$

(c) $\text{AB}_{16} + 3\text{E}_{16}$

(d) $8\text{F}_{16} + \text{AD}_{16}$

Exercise 1.59 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

(a) $22_{16} + 8_{16}$

(b) $73_{16} + 2C_{16}$

(c) $7F_{16} + 7F_{16}$

(d) $C2_{16} + A4_{16}$

Exercise 1.60 Convert the following decimal numbers to 5-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 5-bit result.

(a) $9_{10} - 7_{10}$

(b) $12_{10} - 15_{10}$

(c) $-6_{10} - 11_{10}$

(d) $4_{10} - -8_{10}$

Exercise 1.61 Convert the following decimal numbers to 6-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 6-bit result.

(a) $18_{10} - 12_{10}$

(b) $30_{10} - 9_{10}$

(c) $-28_{10} - 3_{10}$

(d) $-16_{10} -21_{10}$

Exercise 1.62 In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111, 1 as 10000, and so forth. Biased number systems are sometimes used in floating point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}.

(a) What decimal value does the binary number 10000010_2 represent?

(b) What binary number represents the value 0?

(c) What is the representation and value of the most negative number?

(d) What is the representation and value of the most positive number?

Exercise 1.63 Draw a number line analogous to Figure 1.11 for 3-bit biased numbers with a bias of 3 (see Exercise 1.62 for a definition of biased numbers).

Exercise 1.64 In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

(a) Write 289_{10} in BCD

(b) Convert 100101010001_{BCD} to decimal

(c) Convert 01101001_{BCD} to binary

(d) Explain why BCD might be a useful way to represent numbers

Exercise 1.65 Answer the following questions related to BCD systems (see Exercise 1.64 for the definition of BCD).

(a) Write 371_{10} in BCD

(b) Convert 000110000111_{BCD} to decimal

(c) Convert 10010101_{BCD} to binary

(d) Explain the disadvantages of BCD when compared to binary representations of numbers

Exercise 1.66 A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number system: $325 + 42 = 411$. If this equation is correct, how many fingers would you expect Martians to have?

Exercise 1.67 Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, "All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation." Alyssa disagrees. She says, "No, but all such numbers have an even number of 1's in their representation." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.68 Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, "I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result." Alyssa says, "No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.69 Write a program in your favorite language (e.g., C, Java, Perl) to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should print the decimal equivalent.

Exercise 1.70 Repeat Exercise 1.69 but convert from an arbitrary base b_1 to another base b_2, as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9. The user should enter b_1, b_2, and then the number to convert in base b_1. The program should print the equivalent number in base $b_{2.}$

Exercise 1.71 Draw the symbol, Boolean equation, and truth table for

(a) a three-input OR gate

(b) a three-input exclusive OR (XOR) gate

(c) a four-input XNOR gate

Exercise 1.72 Draw the symbol, Boolean equation, and truth table for

(a) a four-input OR gate

(b) a three-input XNOR gate

(c) a five-input NAND gate

Exercise 1.73 A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.41.

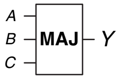


Figure 1.41 Three-input majority gate

Exercise 1.74 A three-input *AND-OR (AO) gate* shown in Figure 1.42 produces a TRUE output if both *A* and *B* are TRUE, or if *C* is TRUE. Complete a truth table for the gate.

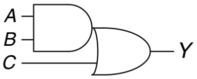


Figure 1.42 Three-input AND-OR gate

Exercise 1.75 A three-input *OR-AND-INVERT (OAI) gate* shown in Figure 1.43 produces a FALSE output if *C* is TRUE and *A* or *B* is TRUE. Otherwise it produces a TRUE output. Complete a truth table for the gate.

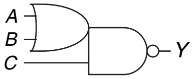


Figure 1.43 **Three-input OR-AND-INVERT gate**

Exercise 1.76 There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

Exercise 1.77 How many different truth tables exist for Boolean functions of N variables?

Exercise 1.78 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.44 would serve as an inverter? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

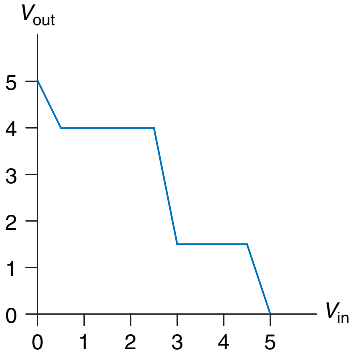


Figure 1.44 **DC transfer characteristics**

Exercise 1.79 Repeat Exercise 1.78 for the transfer characteristics shown in Figure 1.45.

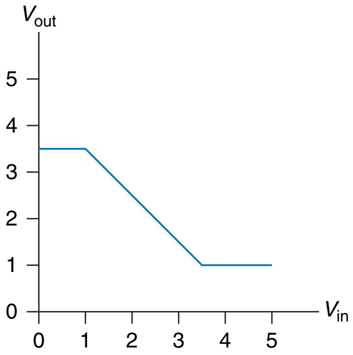


Figure 1.45 **DC transfer characteristics**

Exercise 1.80 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.46 would serve as a buffer? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

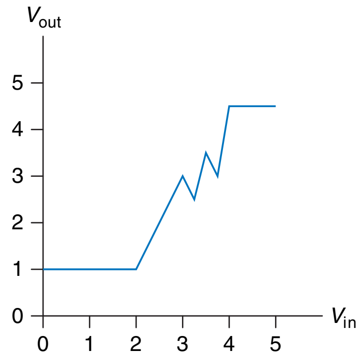


Figure 1.46 DC transfer characteristics

Exercise 1.81 Ben Bitdiddle has invented a circuit with the transfer characteristics shown in Figure 1.47 that he would like to use as a buffer. Will it work? Why or why not? He would like to advertise that it is compatible with LVCMOS and LVTTL logic. Can Ben's buffer correctly receive inputs from those logic families? Can its output properly drive those logic families? Explain.

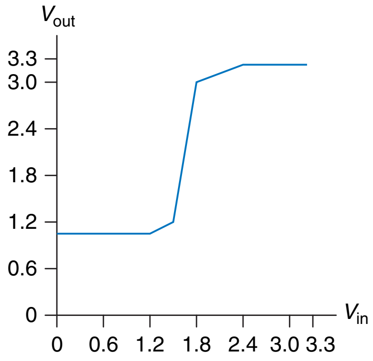


Figure 1.47 Ben's buffer DC transfer characteristics

Exercise 1.82 While walking down a dark alley, Ben Bitdiddle encounters a two-input gate with the transfer function shown in Figure 1.48. The inputs are A and B and the output is Y.

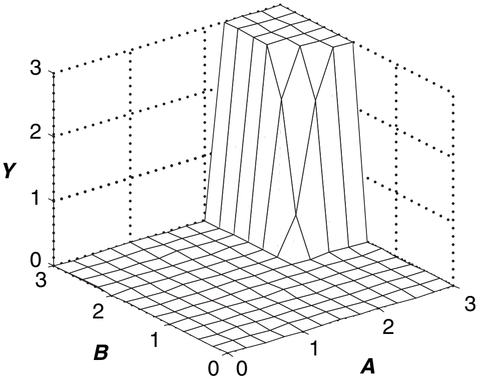


Figure 1.48 Two-input DC transfer characteristics

(a) What kind of logic gate did he find?

(b) What are the approximate high and low logic levels?

Exercise 1.83 Repeat Exercise 1.82 for Figure 1.49.

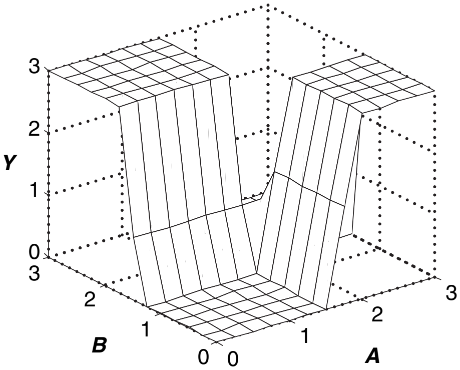


Figure 1.49 Two-input DC transfer characteristics

Exercise 1.84 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) four-input NAND gate

(b) three-input OR-AND-INVERT gate (see Exercise 1.75)

(c) three-input AND-OR gate (see Exercise 1.74)

Exercise 1.85 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input AND gate

(c) two-input OR gate

Exercise 1.86 A *minority gate* produces a TRUE output if and only if fewer than half of its inputs are TRUE. Otherwise it produces a FALSE output. Sketch a transistor-level circuit for a three-input CMOS minority gate. Use a minimum number of transistors.

Exercise 1.87 Write a truth table for the function performed by the gate in Figure 1.50. The truth table should have two inputs, *A* and *B*. What is the name of this function?

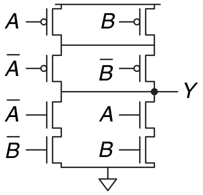


Figure 1.50 **Mystery schematic**

Exercise 1.88 Write a truth table for the function performed by the gate in Figure 1.51. The truth table should have three inputs, *A*, *B*, and C.

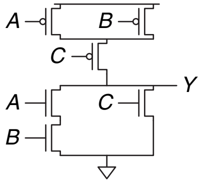


Figure 1.51 **Mystery schematic**

Exercise 1.89 Implement the following three-input gates using only pseudo-nMOS logic gates. Your gates receive three inputs, *A*, *B*, and C. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input NAND gate

(c) three-input AND gate

Exercise 1.90 *Resistor-Transistor Logic* (*RTL*) uses nMOS transistors to pull the gate output LOW and a weak resistor to pull the output HIGH when none of the paths to ground are active. A NOT gate built using RTL is shown in Figure 1.52. Sketch a three-input RTL NOR gate. Use a minimum number of transistors.

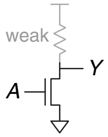


Figure 1.52 **RTL NOT gate**

Interview Questions

These questions have been asked at interviews for digital design jobs.

Question 1.1 Sketch a transistor-level circuit for a CMOS four-input NOR gate.

Question 1.2 The king receives 64 gold coins in taxes but has reason to believe that one is counterfeit. He summons you to identify the fake coin. You have a balance that can hold coins on each side. How many times do you need to use the balance to find the lighter, fake coin?

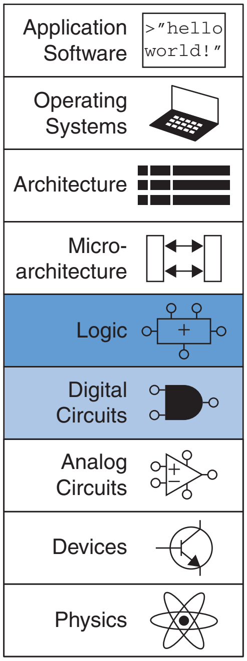
Question 1.3 The professor, the teaching assistant, the digital design student, and the freshman track star need to cross a rickety bridge on a dark night. The bridge is so shaky that only two people can cross at a time. They have only one flashlight among them and the span is too long to throw the flashlight, so somebody must carry it back to the other people. The freshman track star can cross the bridge in 1 minute. The digital design student can cross the bridge in 2 minutes. The teaching assistant can cross the bridge in 5 minutes. The professor always gets distracted and takes 10 minutes to cross the bridge. What is the fastest time to get everyone across the bridge?

This page intentionally left blank

Combinational Logic Design

2

2.1 **Introduction**
2.2 **Boolean Equations**
2.3 **Boolean Algebra**
2.4 **From Logic to Gates**
2.5 **Multilevel Combinational Logic**
2.6 **X's and Z's, Oh My**
2.7 **Karnaugh Maps**
2.8 **Combinational Building Blocks**
2.9 **Timing**
2.10 **Summary**
Exercises
Interview Questions


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
+
Digital Circuits
Analog Circuits
+
−
Devices
Physics

2.1 INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and outputs responding.

Peering inside the black box, circuits are composed of nodes and elements. An *element* is itself a circuit with inputs, outputs, and a specification. A *node* is a wire, whose voltage conveys a discrete-valued variable. Nodes are classified as *input, output*, or *internal*. Inputs receive values from the external world. Outputs deliver values to the external world. Wires that are not inputs or outputs are called internal nodes. Figure 2.2

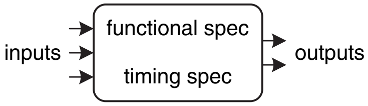

inputs
functional spec
timing spec
outputs

Figure 2.1 **Circuit as a black box with inputs, outputs, and specifications**

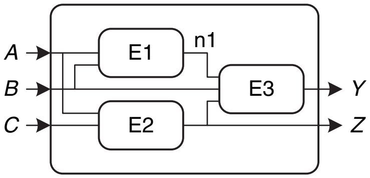
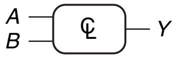

A
B
C
E1
n1
E2
E3
Y
Z

Figure 2.2 **Elements and nodes**

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00002-1
© 2013 Elsevier, Inc. All rights reserved.

illustrates a circuit with three elements, E1, E2, and E3, and six nodes. Nodes A, B, and C are inputs. Y and Z are outputs. n1 is an internal node between E1 and E3.

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence. A combinational circuit is *memoryless*, but a sequential circuit has *memory*. This chapter focuses on combinational circuits, and Chapter 3 examines sequential circuits.



$Y = F(A, B) = A + B$

Figure 2.3 **Combinational logic circuit**

The functional specification of a combinational circuit expresses the output values in terms of the current input values. The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. We will initially concentrate on the functional specification, then return to the timing specification later in this chapter.

Figure 2.3 shows a combinational circuit with two inputs and one output. On the left of the figure are the inputs, A and B, and on the right is the output, Y. The symbol CL inside the box indicates that it is implemented using only combinational logic. In this example, the function F is specified to be OR: $Y = F(A, B) = A + B$. In words, we say the output Y is a function of the two inputs, A and B, namely $Y = A$ OR B.

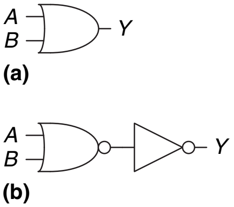
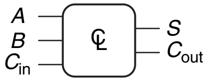


Figure 2.4 **Two OR implementations**

Figure 2.4 shows two possible *implementations* for the combinational logic circuit in Figure 2.3. As we will see repeatedly throughout the book, there are often many implementations for a single function. You choose which to use given the building blocks at your disposal and your design constraints. These constraints often include area, speed, power, and design time.



$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2.5 **Multiple-output combinational circuit**

Figure 2.5 shows a combinational circuit with multiple outputs. This particular combinational circuit is called a *full adder* and we will revisit it in Section 5.2.1. The two equations specify the function of the outputs, S and C_{out}, in terms of the inputs, A, B, and C_{in}.

To simplify drawings, we often use a single line with a slash through it and a number next to it to indicate a *bus*, a bundle of multiple signals. The number specifies how many signals are in the bus. For example, Figure 2.6(a) represents a block of combinational logic with three inputs and two outputs. If the number of bits is unimportant or obvious from the context, the slash may be shown without a number. Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.

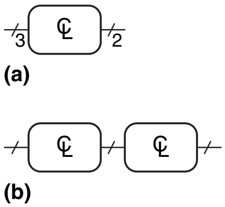


Figure 2.6 **Slash notation for multiple signals**

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit elements.

A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

The rules of combinational composition are sufficient but not strictly necessary. Certain circuits that disobey these rules are still combinational, so long as the outputs depend only on the current values of the inputs. However, determining whether oddball circuits are combinational is more difficult, so we will usually restrict ourselves to combinational composition as a way to build combinational circuits.

Example 2.1 COMBINATIONAL CIRCUITS

Which of the circuits in Figure 2.7 are combinational circuits according to the rules of combinational composition?

Solution: Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

Large circuits such as microprocessors can be very complicated, so we use the principles from Chapter 1 to manage the complexity. Viewing a circuit as a black box with a well-defined interface and function is an application of abstraction and modularity. Building the circuit out of

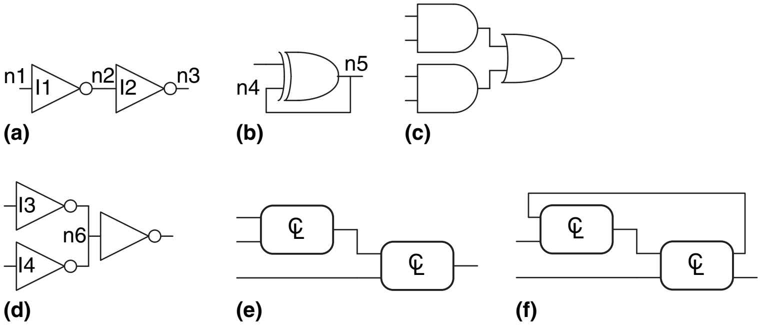


Figure 2.7 Example circuits

smaller circuit elements is an application of hierarchy. The rules of combinational composition are an application of discipline.

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations. We show how to implement these equations using logic gates and how to analyze the speed of these circuits.

2.2 BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic. This section defines some terminology commonly used in Boolean equations, then shows how to write a Boolean equation for any logic function given its truth table.

2.2.1 Terminology

The *complement* of a variable A is its inverse $\overline{A}$. The variable or its complement is called a *literal*. For example, A, $\overline{A}$, B, and $\overline{B}$ are literals. We call A the *true form* of the variable and $\overline{A}$ the *complementary form;* "true form" does not mean that A is TRUE, but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant*. $\overline{A}B$, $A\overline{B}\,\overline{C}$, and B are all implicants for a function of three variables. A *minterm* is a product involving all of the inputs to the function. $A\overline{B}\,\overline{C}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not, because it does not involve C. Similarly, the OR of one or more literals is called a *sum*. A *maxterm* is a sum involving all of the inputs to the function. $A+\overline{B}+C$ is a maxterm for a function of the three variables A, B, and C.

The *order of operations* is important when interpreting Boolean equations. Does $Y = A + BC$ mean $Y = (A$ OR $B)$ AND C or $Y = A$ OR $(B$ AND $C)$? In Boolean equations, NOT has the highest *precedence*, followed by AND, then OR. Just as in ordinary equations, products are performed before sums. Therefore, the equation is read as $Y = A$ OR $(B$ AND $C)$. Equation 2.1 gives another example of order of operations.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \tag{2.1}$$

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	0	$A\ B$	m_3

Figure 2.8 Truth table and minterms

2.2.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs. Each row in a truth table is associated with a minterm that is TRUE for that row. Figure 2.8 shows a truth table of two inputs, A and B. Each row shows its corresponding minterm. For example, the minterm for the first row is $\overline{A}\,\overline{B}$ because $\overline{A}\,\overline{B}$ is TRUE when $A = 0$, $B = 0$. The minterms are numbered starting with 0; the top row corresponds to minterm 0, m_0, the next row to minterm 1, m_1, and so on.

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. For example, in Figure 2.8, there is only one row (or minterm) for which the output Y is TRUE, shown circled in blue. Thus, $Y = \overline{A}B$. Figure 2.9 shows a truth table with more than one row in which the output is TRUE. Taking the sum of each of the circled minterms gives $Y = \overline{A}B + AB$.

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

Figure 2.9 **Truth table with multiple TRUE minterms**

This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms). Although there are many ways to write the same function, such as $Y = B\overline{A} + BA$, we will sort the minterms in the same order that they appear in the truth table, so that we always write the same Boolean expression for the same truth table.

Canonical form is just a fancy word for standard form. You can use the term to impress your friends and scare your enemies.

The sum-of-products canonical form can also be written in *sigma notation* using the summation symbol, Σ. With this notation, the function from Figure 2.9 would be written as:

$$F(A,B) = \Sigma(m_1, m_3)$$

or (2.2)

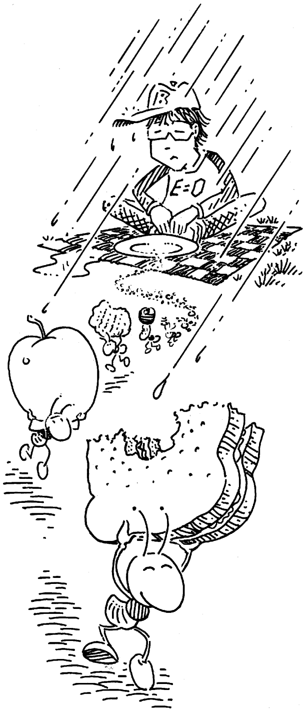
$$F(A,B) = \Sigma(1,3)$$

Example 2.2 SUM-OF-PRODUCTS FORM

Ben Bitdiddle is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE *only* if Ben enjoys the picnic.

Solution: First define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers. Figure 2.10 shows the truth table for Ben's picnic experience.

Using sum-of-products form, we write the equation as: $E = \overline{A}\,\overline{R}$ or $E = \Sigma(0)$. We can build the equation using two inverters and a two-input AND gate, shown in Figure 2.11(a). You may recognize this truth table as the NOR function from Section 1.5.5: $E = A$ NOR R $= \overline{A + R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}\,\overline{R}$ and $\overline{A + R}$, are equivalent.



A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.10 **Ben's truth table**

The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Figure 2.12 shows a random three-input truth table. The sum-of-products form of the logic function is

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

or (2.3)

$$Y = \Sigma(0, 4, 5)$$

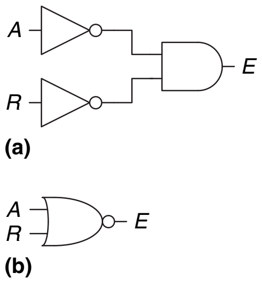


Figure 2.11 **Ben's circuit**

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.12 **Random three-input truth table**

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Figure 2.13 **Truth table with multiple FALSE maxterms**

Unfortunately, sum-of-products form does not necessarily generate the simplest equation. In Section 2.3 we show how to write the same function using fewer terms.

2.2.3 Product-of-Sums Form

An alternative way of expressing Boolean functions is the *product-of-sums canonical form*. Each row of a truth table corresponds to a maxterm that is FALSE for that row. For example, the maxterm for the first row of a two-input truth table is $(A + B)$ because $(A + B)$ is FALSE when $A = 0$, $B = 0$. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE. The product-of-sums canonical form can also be written in *pi notation* using the product symbol, Π.

Example 2.3 PRODUCT-OF-SUMS FORM

Write an equation in product-of-sums form for the truth table in Figure 2.13.

Solution: The truth table has two rows in which the output is FALSE. Hence, the function can be written in product-of-sums form as $Y = (A + B)(\overline{A} + B)$ or, using pi notation, $Y = \Pi(M_0, M_2)$ or $Y = \Pi(0, 2)$. The first maxterm, $(A + B)$, guarantees that $Y = 0$ for $A = 0$, $B = 0$, because any value AND 0 is 0. Likewise, the second maxterm, $(\overline{A} + \mathrm{B})$, guarantees that $Y = 0$ for $A = 1$, $B = 0$. Figure 2.13 is the same truth table as Figure 2.9, showing that the same function can be written in more than one way.

Similarly, a Boolean equation for Ben's picnic from Figure 2.10 can be written in product-of-sums form by circling the three rows of 0's to obtain $E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$ or $E = \Pi(1, 2, 3)$. This is uglier than the sum-of-products equation, $E = \overline{A}\,\overline{R}$, but the two equations are logically equivalent.

Sum-of-products produces a shorter equation when the output is TRUE on only a few rows of a truth table; product-of-sums is simpler when the output is FALSE on only a few rows of a truth table.

2.3 BOOLEAN ALGEBRA

In the previous section, we learned how to write a Boolean expression given a truth table. However, that expression does not necessarily lead to the simplest set of logic gates. Just as you use algebra to simplify mathematical equations, you can use *Boolean algebra* to simplify Boolean equations. The rules of Boolean algebra are much like those of ordinary algebra but are in some cases simpler, because variables have only two possible values: 0 or 1.

Boolean algebra is based on a set of axioms that we assume are correct. Axioms are unprovable in the sense that a definition cannot be proved. From these axioms, we prove all the theorems of Boolean algebra.

Table 2.1 **Axioms of Boolean algebra**

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

These theorems have great practical significance, because they teach us how to simplify logic to produce smaller and less costly circuits.

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators • (AND) and + (OR) are interchanged, the statement will still be correct. We use the prime symbol (′) to denote the *dual* of a statement.

2.3.1 Axioms

Table 2.1 states the axioms of Boolean algebra. These five axioms and their duals define Boolean variables and the meanings of NOT, AND, and OR. Axiom A1 states that a Boolean variable *B* is 0 if it is not 1. The axiom's dual, A1′, states that the variable is 1 if it is not 0. Together, A1 and A1′ tell us that we are working in a Boolean or binary field of 0's and 1's. Axioms A2 and A2′ define the NOT operation. Axioms A3 to A5 define AND; their duals, A3′ to A5′ define OR.

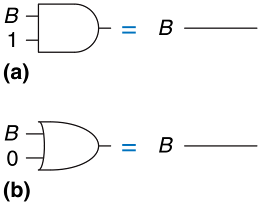


Figure 2.14 **Identity theorem in hardware: (a) T1, (b) T1′**

2.3.2 Theorems of One Variable

Theorems T1 to T5 in Table 2.2 describe how to simplify equations involving one variable.

The *identity* theorem, T1, states that for any Boolean variable *B*, *B* AND 1 = *B*. Its dual states that *B* OR 0 = *B*. In hardware, as shown in Figure 2.14, T1 means that if one input of a two-input AND gate is always 1, we can remove the AND gate and replace it with a wire connected to the variable input (*B*). Likewise, T1′ means that if one input of a two-input OR gate is always 0, we can replace the OR gate with a wire connected to *B*. In general, gates cost money, power, and delay, so replacing a gate with a wire is beneficial.

The *null element theorem*, T2, says that *B* AND 0 is always equal to 0. Therefore, 0 is called the *null* element for the AND operation, because it nullifies the effect of any other input. The dual states that *B* OR 1 is always equal to 1. Hence, 1 is the null element for the OR operation. In hardware,

The null element theorem leads to some outlandish statements that are actually true! It is particularly dangerous when left in the hands of advertisers: YOU WILL GET A MILLION DOLLARS or we'll send you a toothbrush in the mail. (You'll most likely be receiving a toothbrush in the mail.)

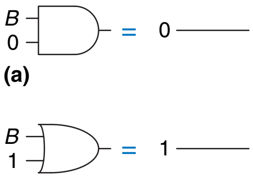


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

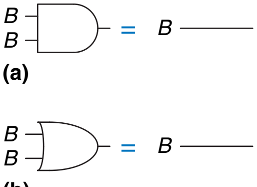


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

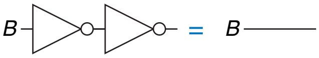


Figure 2.17 Involution theorem in hardware: T4

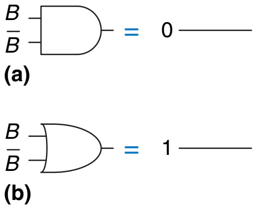


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\bar{\bar{B}} = B$		Involution
T5	$B \bullet \bar{B} = 0$	T5′	$B + \bar{B} = 1$	Complements

as shown in Figure 2.15, if one input of an AND gate is 0, we can replace the AND gate with a wire that is tied LOW (to 0). Likewise, if one input of an OR gate is 1, we can replace the OR gate with a wire that is tied HIGH (to 1).

Idempotency, T3, says that a variable AND itself is equal to just itself. Likewise, a variable OR itself is equal to itself. The theorem gets its name from the Latin roots: *idem* (same) and *potent* (power). The operations return the same thing you put into them. Figure 2.16 shows that idempotency again permits replacing a gate with a wire.

Involution, T4, is a fancy way of saying that complementing a variable twice results in the original variable. In digital electronics, two wrongs make a right. Two inverters in series logically cancel each other out and are logically equivalent to a wire, as shown in Figure 2.17. The dual of T4 is itself.

The *complement theorem*, T5 (Figure 2.18), states that a variable AND its complement is 0 (because one of them has to be 0). And by duality, a variable OR its complement is 1 (because one of them has to be 1).

2.3.3 Theorems of Several Variables

Theorems T6 to T12 in Table 2.3 describe how to simplify equations involving more than one Boolean variable.

Commutativity and *associativity*, T6 and T7, work the same as in traditional algebra. By commutativity, the *order* of inputs for an AND or OR function does not affect the value of the output. By associativity, the specific groupings of inputs do not affect the value of the output.

The *distributivity theorem*, T8, is the same as in traditional algebra, but its dual, T8′, is not. By T8, AND distributes over OR, and by T8′, OR distributes over AND. In traditional algebra, multiplication distributes over addition but addition does not distribute over multiplication, so that $(B + C) \times (B + D) \neq B + (C \times D)$.

The *covering*, *combining*, and *consensus* theorems, T9 to T11, permit us to eliminate redundant variables. With some thought, you should be able to convince yourself that these theorems are correct.

Table 2.3 Boolean theorems of several variables

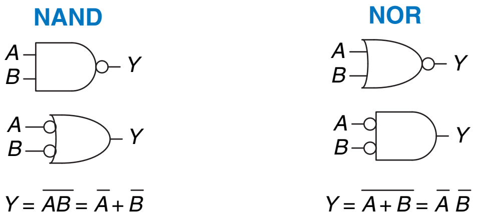
	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots)$	De Morgan's Theorem

De Morgan's Theorem, T12, is a particularly powerful tool in digital design. The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

According to De Morgan's theorem, a NAND gate is equivalent to an OR gate with inverted inputs. Similarly, a NOR gate is equivalent to an AND gate with inverted inputs. Figure 2.19 shows these *De Morgan equivalent gates* for NAND and NOR gates. The two symbols shown for each function are called *duals*. They are logically equivalent and can be used interchangeably.

The inversion circle is called a *bubble*. Intuitively, you can imagine that "pushing" a bubble through the gate causes it to come out at the other side

Augustus De Morgan, died 1871. A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2."


NAND
A
B
Y
A
B
Y
$Y = \overline{AB} = \overline{A} + \overline{B}$
NOR
A
B
Y
A
B
Y
$Y = \overline{A + B} = \overline{A}\,\overline{B}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.19 De Morgan equivalent gates

and flips the body of the gate from AND to OR or vice versa. For example, the NAND gate in Figure 2.19 consists of an AND body with a bubble on the output. Pushing the bubble to the left results in an OR body with bubbles on the inputs. The underlying rules for bubble pushing are

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on *all* gate inputs forward toward the output puts a bubble on the output.

Section 2.5.2 uses bubble pushing to help analyze circuits.

Example 2.4 DERIVE THE PRODUCT-OF-SUMS FORM

Figure 2.20 shows the truth table for a Boolean function Y and its complement $\overline{Y}$. Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Figure 2.20 Truth table showing *Y* and $\overline{Y}$

Solution: Figure 2.21 shows the minterms (circled) contained in $\overline{Y}$. The sum-of-products canonical form of $\overline{Y}$ is

A	B	Y	$\overline{Y}$	minterm
0	0	0	**1**	$\overline{A}\,\overline{B}$
0	1	0	**1**	$\overline{A}\,B$
1	0	1	**0**	$A\,\overline{B}$
1	1	1	**0**	$A\,B$

Figure 2.21 Truth table showing minterms for $\overline{Y}$

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B \tag{2.4}$$

Taking the complement of both sides and applying De Morgan's Theorem twice, we get:

$$\overline{\overline{Y}} = Y = \overline{\overline{A}\,\overline{B} + \overline{A}B} = (\overline{\overline{A}\,\overline{B}})(\overline{\overline{A}B}) = (A+B)(A+\overline{B}) \tag{2.5}$$

2.3.4 The Truth Behind It All

The curious reader might wonder how to prove that a theorem is true. In Boolean algebra, proofs of theorems with a finite number of variables are easy: just show that the theorem holds for all possible values of these variables. This method is called *perfect induction* and can be done with a truth table.

Example 2.5 PROVING THE CONSENSUS THEOREM USING PERFECT INDUCTION

Prove the consensus theorem, T11, from Table 2.3.

Solution: Check both sides of the equation for all eight combinations of B, C, and D. The truth table in Figure 2.22 illustrates these combinations. Because $BC + \overline{B}D + CD = BC + \overline{B}D$ for all cases, the theorem is proved.

B	C	D	$BC+\overline{B}D+CD$	$BC+\overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Figure 2.22 **Truth table proving T11**

2.3.5 Simplifying Equations

The theorems of Boolean algebra help us simplify Boolean equations. For example, consider the sum-of-products expression from the truth table of Figure 2.9: $Y=\overline{A}B+AB$. By Theorem T10, the equation simplifies to $Y=B$. This may have been obvious looking at the truth table. In general, multiple steps may be necessary to simplify more complex equations.

The basic principle of simplifying sum-of-products equations is to combine terms using the relationship $PA+P\overline{A}=P$, where P may be any implicant. How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants in the equation to form a new implicant with fewer literals. The implicants in a minimal equation must all be prime implicants. Otherwise, they could be combined to reduce the number of literals.

Example 2.6 EQUATION MINIMIZATION

Minimize Equation 2.3: $\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$.

Solution: We start with the original equation and apply Boolean theorems step by step, as shown in Table 2.4.

Have we simplified the equation completely at this point? Let's take a closer look. From the original equation, the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $A\overline{B}\,\overline{C}$ differ only in the variable A. So we combined the minterms to form $\overline{B}\,\overline{C}$. However, if we look at the original equation, we note that the last two minterms $A\overline{B}\,\overline{C}$ and $A\overline{B}C$ also differ by a single literal (C and $\overline{C}$). Thus, using the same method, we could have combined these two minterms to form the minterm $A\overline{B}$. We say that implicants $\overline{B}\,\overline{C}$ and $A\overline{B}$ *share* the minterm $A\overline{B}\,\overline{C}$.

So, are we stuck with simplifying only one of the minterm pairs, or can we simplify both? Using the idempotency theorem, we can duplicate terms as many times as we want: $B=B+B+B+B\ldots$. Using this principle, we simplify the equation completely to its two prime implicants, $\overline{B}\,\overline{C}+A\overline{B}$, as shown in Table 2.5.

Table 2.4 **Equation minimization**

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}C$	T8: Distributivity
2	$\overline{B}\,\overline{C}(1) + A\overline{B}C$	T5: Complements
3	$\overline{B}\,\overline{C} + A\overline{B}C$	T1: Identity

Table 2.5 **Improved equation minimization**

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1) + A\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C} + A\overline{B}$	T1: Identity

Although it is a bit counterintuitive, *expanding* an implicant (for example, turning AB into $ABC + AB\overline{C}$) is sometimes useful in minimizing equations. By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called Karnaugh maps that makes the process easier.

Why bother simplifying a Boolean equation if it remains logically equivalent? Simplifying reduces the number of gates used to physically implement the function, thus making it smaller, cheaper, and possibly faster. The next section describes how to implement Boolean equations with logic gates.

The labs that accompany this textbook (see Preface) show how to use *computer-aided design* (CAD) tools to design, simulate, and test digital circuits.

2.4 FROM LOGIC TO GATES

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together. For example, the schematic in Figure 2.23 shows a possible hardware implementation of our favorite logic function, Equation 2.3:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

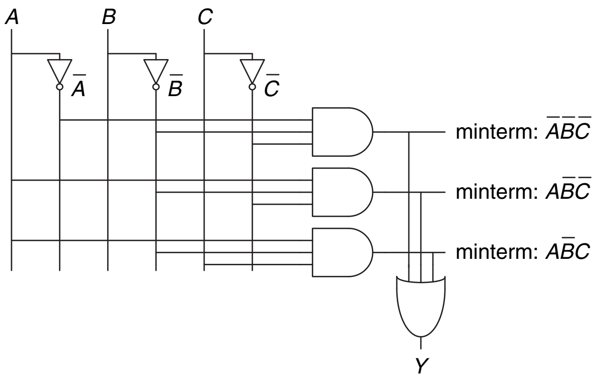

A
B
C
$\overline{A}$
$\overline{B}$
$\overline{C}$
minterm: $\overline{A}\overline{B}\overline{C}$
minterm: $A\overline{B}\overline{C}$
minterm: $A\overline{B}C$
Y

Figure 2.23 Schematic of $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking of what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

The last three guidelines are illustrated in Figure 2.24.

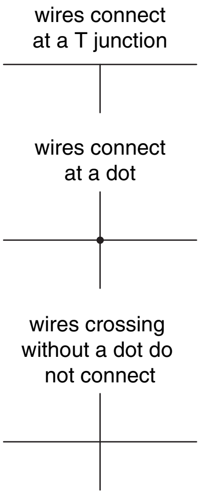

wires connect
at a T junction
wires connect
at a dot
wires crossing
without a dot do
not connect

Figure 2.24 Wire connections

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to Figure 2.23. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This style is called a *programmable logic array* (*PLA*) because the inverters, AND gates, and OR gates are arrayed in a systematic fashion. PLAs will be discussed further in Section 5.6.

Figure 2.25 shows an implementation of the simplified equation we found using Boolean algebra in Example 2.6. Notice that the simplified circuit has significantly less hardware than that of Figure 2.23. It may also be faster, because it uses gates with fewer inputs.

We can reduce the number of gates even further (albeit by a single inverter) by taking advantage of inverting gates. Observe that $\overline{B}\,\overline{C}$ is an

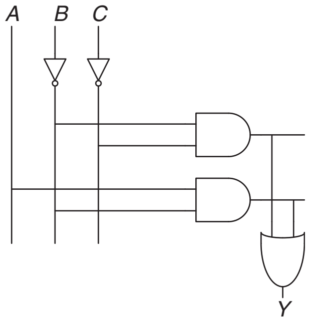


Figure 2.25 Schematic of $Y = \overline{B}\,\overline{C} + A\overline{B}$

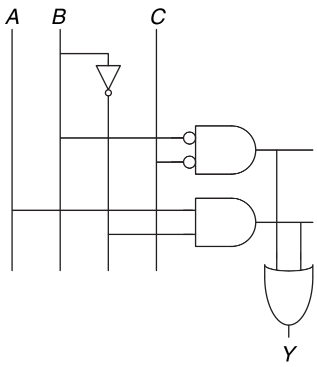


Figure 2.26 **Schematic using fewer gates**

AND with inverted inputs. Figure 2.26 shows a schematic using this optimization to eliminate the inverter on C. Recall that by De Morgan's theorem the AND with inverted inputs is equivalent to a NOR gate. Depending on the implementation technology, it may be cheaper to use the fewest gates or to use certain types of gates in preference to others. For example, NANDs and NORs are preferred over ANDs and ORs in CMOS implementations.

Many circuits have multiple outputs, each of which computes a separate Boolean function of the inputs. We can write a separate truth table for each output, but it is often convenient to write all of the outputs on a single truth table and sketch one schematic with all of the outputs.

Example 2.7 MULTIPLE-OUTPUT CIRCUITS

The dean, the department chair, the teaching assistant, and the dorm social chair each use the auditorium from time to time. Unfortunately, they occasionally conflict, leading to disasters such as the one that occurred when the dean's fundraising meeting with crusty trustees happened at the same time as the dorm's BTB[1] party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, $A_3, \ldots, A_0$, and four outputs, $Y_3, \ldots, Y_0$. These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts her input when she requests the auditorium for the next day. The system asserts at most one output, granting the auditorium to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority.

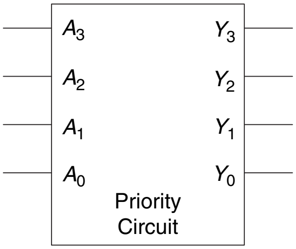
Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.

Solution: This function is called a four-input *priority circuit*. Its symbol and truth table are shown in Figure 2.27.

We could write each output in sum-of-products form and reduce the equations using Boolean algebra. However, the simplified equations are clear by inspection from the functional description (and the truth table): Y_3 is TRUE whenever A_3 is asserted, so $Y_3 = A_3$. Y_2 is TRUE if A_2 is asserted and A_3 is not asserted, so $Y_2 = \overline{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher priority inputs is asserted: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$. And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$. The schematic is shown in Figure 2.28. An experienced designer can often implement a logic circuit by inspection. Given a clear specification, simply turn the words into equations and the equations into gates.

---

[1] Black light, twinkies, and beer.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2.27 **Priority circuit**

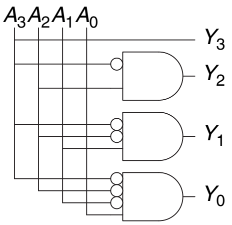


Figure 2.28 **Priority circuit schematic**

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 **Priority circuit truth table with don't cares (X's)**

Notice that if A_3 is asserted in the priority circuit, the outputs *don't care* what the other inputs are. We use the symbol X to describe inputs that the output doesn't care about. Figure 2.29 shows that the four-input priority circuit truth table becomes much smaller with don't cares. From this truth table, we can easily read the Boolean equations in sum-of-products form by ignoring inputs with X's. Don't cares can also appear in truth table outputs, as we will see in Section 2.7.3.

X is an overloaded symbol that means "don't care" in truth tables and "contention" in logic simulation (see Section 2.6.1). Think about the context so you don't mix up the meanings. Some authors use D or ? instead for "don't care" to avoid this ambiguity.

2.5 MULTILEVEL COMBINATIONAL LOGIC

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts. Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

2.5.1 Hardware Reduction

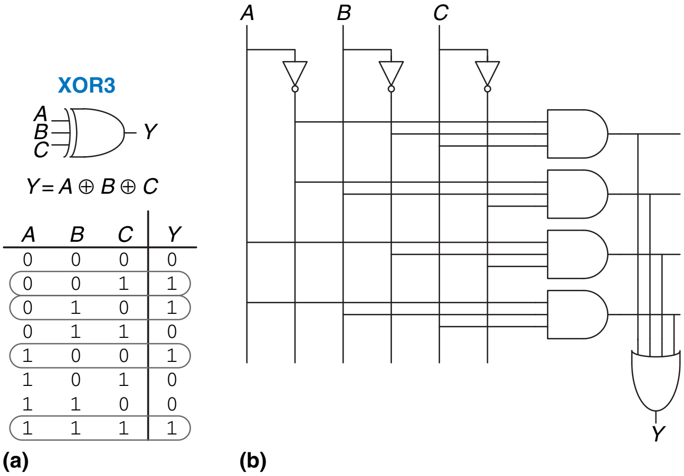
Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. For example, consider building a three-input XOR using the two-level techniques we have studied so far.

Recall that an *N*-input XOR produces a TRUE output when an odd number of inputs are TRUE. Figure 2.30 shows the truth table for a three-input XOR with the rows circled that produce TRUE outputs. From the truth table, we read off a Boolean equation in sum-of-products form in Equation 2.6. Unfortunately, there is no way to simplify this equation into fewer implicants.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{2.6}$$

On the other hand, $A \oplus B \oplus C = (A \oplus B) \oplus C$ (prove this to yourself by perfect induction if you are in doubt). Therefore, the three-input XOR can be built out of a cascade of two-input XORs, as shown in Figure 2.31.

Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two-level sum-of-products implementation. A much better option is to use a tree of two-input XOR gates, as shown in Figure 2.32.



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.30 **Three-input XOR: (a) functional specification and (b) two-level logic implementation**

Selecting the best multilevel implementation of a specific logic function is not a simple process. Moreover, "best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption. In Chapter 5, you will see that the "best" circuit in one technology is not necessarily the best in another. For example, we have been using ANDs and ORs, but in CMOS, NANDs and NORs are more efficient. With some experience, you will find that you can create a good multilevel design by inspection for most circuits. You will develop some of this experience as you study circuit examples through the rest of this book. As you are learning, explore various design options and think about the trade-offs. Computer-aided design (CAD) tools are also available to search a vast space of possible multilevel designs and seek the one that best fits your constraints given the available building blocks.

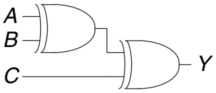


Figure 2.31 Three-input XOR using two-input XORs

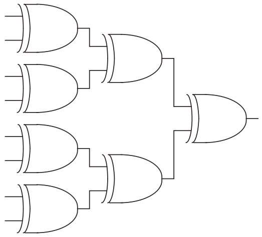


Figure 2.32 Eight-input XOR using seven two-input XORs

2.5.2 Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits prefer NANDs and NORs over ANDs and ORs. But reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Figure 2.33 shows a multilevel circuit whose function is not immediately clear by inspection. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be determined. Building on the principles from Section 2.3.3, the guidelines for bubble pushing are as follows:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (for example, Y) instead of the complement of the output ($\overline{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

Figure 2.34 shows how to redraw Figure 2.33 according to the bubble pushing guidelines. Starting at the output Y, the NAND gate has a bubble on the output that we wish to eliminate. We push the output bubble back to form an OR with inverted inputs, shown in

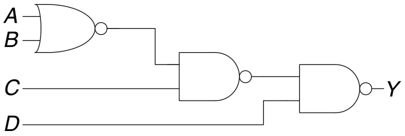


Figure 2.33 Multilevel circuit using NANDs and NORs

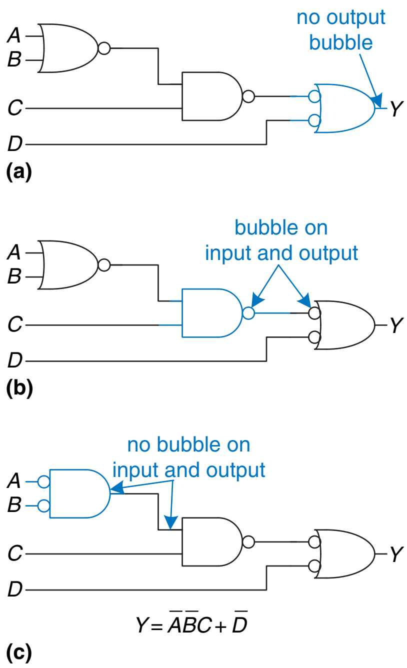

no output
bubble
A
B
C
D
Y
(a)
bubble on
input and output
A
B
C
D
Y
(b)
no bubble on
input and output
A
B
C
D
Y
$Y = \overline{A}\overline{B}C + \overline{D}$
(c)

Figure 2.34 **Bubble-pushed circuit**

Figure 2.34(a). Working to the left, the rightmost gate has an input bubble that cancels with the output bubble of the middle NAND gate, so no change is necessary, as shown in Figure 2.34(b). The middle gate has no input bubble, so we transform the leftmost gate to have no output bubble, as shown in Figure 2.34(c). Now all of the bubbles in the circuit cancel except at the inputs, so the function can be read by inspection in terms of ANDs and ORs of true or complementary inputs: $Y = \overline{A}\,\overline{B}C + \overline{D}$.

For emphasis of this last point, Figure 2.35 shows a circuit logically equivalent to the one in Figure 2.34. The functions of internal nodes are labeled in blue. Because bubbles in series cancel, we can ignore the bubbles on the output of the middle gate and on one input of the rightmost gate to produce the logically equivalent circuit of Figure 2.35.

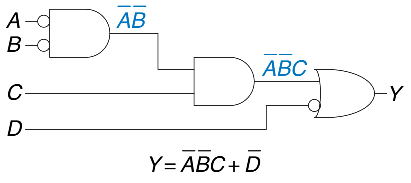

A
B
$\overline{A}\overline{B}$
$\overline{A}\overline{B}C$
C
D
Y
$Y = \overline{A}\overline{B}C + \overline{D}$

Figure 2.35 **Logically equivalent bubble-pushed circuit**

Example 2.8 BUBBLE PUSHING FOR CMOS LOGIC

Most designers think in terms of AND and OR gates, but suppose you would like to implement the circuit in Figure 2.36 in CMOS logic, which favors NAND and NOR gates. Use bubble pushing to convert the circuit to NANDs, NORs, and inverters.

Solution: A brute force solution is to just replace each AND gate with a NAND and an inverter, and each OR gate with a NOR and an inverter, as shown in Figure 2.37. This requires eight gates. Notice that the inverter is drawn with the bubble on the front rather than back, to emphasize how the bubble can cancel with the preceding inverting gate.

For a better solution, observe that bubbles can be added to the output of a gate and the input of the next gate without changing the function, as shown in Figure 2.38(a). The final AND is converted to a NAND and an inverter, as shown in Figure 2.38(b). This solution requires only five gates.

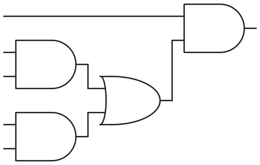


Figure 2.36 **Circuit using ANDs and ORs**

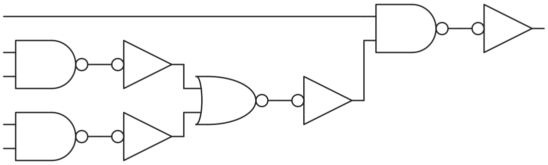
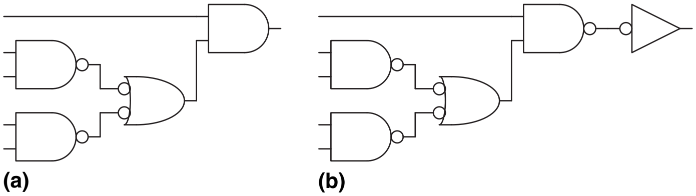


Figure 2.37 **Poor circuit using NANDs and NORs**



(a)

(b)

Figure 2.38 **Better circuit using NANDs and NORs**

2.6 X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

2.6.1 Illegal Value: X

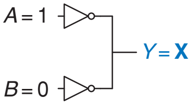


Figure 2.39 **Circuit with contention**

The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.39 shows a case where node Y is driven both HIGH and LOW. This situation, called *contention*, is considered to be an error and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone. Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.

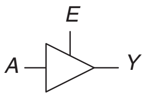
X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume it is an X to warn you of the problem.

As mentioned in Section 2.4, digital designers also use the symbol X to indicate "don't care" values in truth tables. Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant (can be either 0 or 1). When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

2.6.2 Floating Value: Z

The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high* Z. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system. A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.

Tristate Buffer



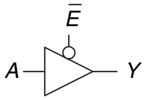
E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer

One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.

The *tristate buffer*, shown in Figure 2.40, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input A, output Y, and *enable* E. When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

The tristate buffer in Figure 2.40 has an *active high* enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure 2.41 shows a tristate buffer with an *active low* enable. When the enable is LOW (0),



$\overline{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

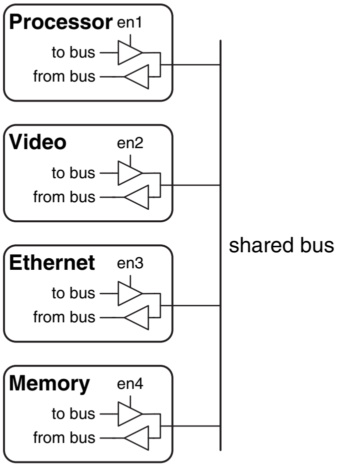


Figure 2.42 **Tristate bus connecting multiple chips**

the buffer is enabled. We show that the signal is active low by putting a bubble on its input wire. We often indicate an active low input by drawing a bar over its name, $\overline{E}$, or appending the letters "b" or "bar" after its name, *Eb* or *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in Figure 2.42. Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time. Such tristate busses were once common. However, in modern computers, higher speeds are possible with *point-to-point links*, in which chips are connected to each other directly rather than over a shared bus.

2.7 KARNAUGH MAPS

After working through several minimizations of Boolean equations using Boolean algebra, you will realize that, if you're not careful, you sometimes end up with a completely *different* equation instead of a simplified equation. *Karnaugh maps* (*K-maps*) are a graphical method for simplifying Boolean equations. They were invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs. K-maps work well for problems

Maurice Karnaugh, 1924–. Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952.

Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

with up to four variables. More important, they give insight into manipulating Boolean equations.

Recall that logic minimization involves combining terms. Two terms containing an implicant P and the true and complementary forms of some variable A are combined to eliminate A: $PA + P\overline{A} = P$. Karnaugh maps make these combinable terms easy to see by putting them next to each other in a grid.

Figure 2.43 shows the truth table and K-map for a three-input function. The top row of the K-map gives the four possible values for the A and B inputs. The left column gives the two possible values for the C input. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row. For example, the top left square corresponds to the first row in the truth table and indicates that the output value $Y = 1$ when $ABC = 000$. Just like each row in a truth table, each square in a K-map represents a single minterm. For the purpose of explanation, Figure 2.43(c) shows the minterm corresponding to each square in the K-map.

Each square, or minterm, differs from an adjacent square by a change in a single variable. This means that adjacent squares share all the same literals except one, which appears in true form in one square and in complementary form in the other. For example, the squares representing the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$ are adjacent and differ only in the variable C. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10. This order is called a *Gray code*. It differs from ordinary binary order (00, 01, 10, 11) in that adjacent entries differ only in a single variable. For example, 01 : 11 only changes A from 0 to 1, while 01 : 10 would change A from 0 to 1 and B from 1 to 0. Hence, writing the combinations in binary order would not have produced our desired property of adjacent squares differing only in one variable.

The K-map also "wraps around." The squares on the far right are effectively adjacent to the squares on the far left, in that they differ only in one variable, A. In other words, you could take the map and roll it into a cylinder, then join the ends of the cylinder to form a torus (i.e., a donut), and still guarantee that adjacent squares would differ only in one variable.

Gray codes were patented (U.S. Patent 2,632,058) by Frank Gray, a Bell Labs researcher, in 1953. They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit.

Gray codes generalize to any number of bits. For example, a 3-bit Gray code sequence is:

`000, 001, 011, 010, 110, 111, 101, 100`

Lewis Carroll posed a related puzzle in *Vanity Fair* in 1879.

> "The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the puzzle consists of linking these together by interposing other words, each of which shall differ from the next word in one letter only. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word."

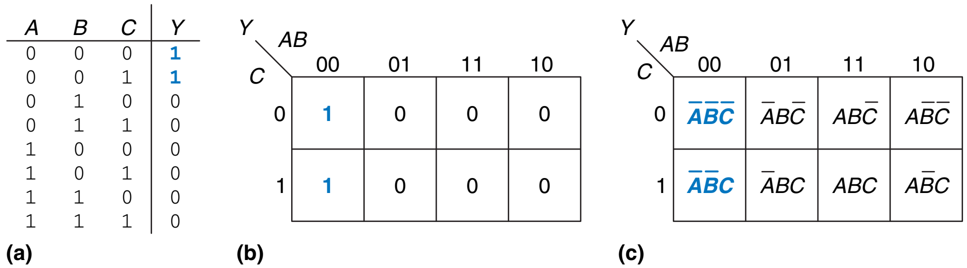
For example, SHIP to DOCK:

`SHIP, SLIP, SLOP, SLOT, SOOT, LOOT, LOOK, LOCK, DOCK.`

Can you find a shorter sequence?

2.7.1 Circular Thinking

In the K-map in Figure 2.43, only two minterms are present in the equation, $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$, as indicated by the 1's in the left column. Reading the minterms from the K-map is exactly equivalent to reading equations in sum-of-products form directly from the truth table.



A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms

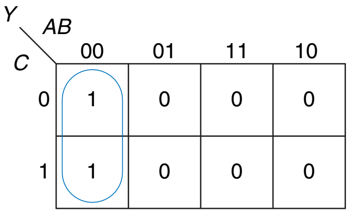


Figure 2.44 K-map minimization

As before, we can use Boolean algebra to minimize equations in sum-of-products form.

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C = \overline{A}\,\overline{B}(\overline{C} + C) = \overline{A}\,\overline{B} \tag{2.7}$$

K-maps help us do this simplification graphically by *circling* 1's in adjacent squares, as shown in Figure 2.44. For each circle, we write the corresponding implicant. Remember from Section 2.2 that an implicant is the product of one or more literals. Variables whose true *and* complementary forms are both in the circle are excluded from the implicant. In this case, the variable C has both its true form (1) and its complementary form (0) in the circle, so we do not include it in the implicant. In other words, Y is TRUE when $A = B = 0$, independent of C. So the implicant is $\overline{A}\,\overline{B}$. The K-map gives the same answer we reached using Boolean algebra.

2.7.2 Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as large as possible. Then read off the implicants that were circled.

More formally, recall that a Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants. Each circle on the K-map represents an implicant. The largest possible circles are prime implicants.

For example, in the K-map of Figure 2.44, $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$ are implicants, but *not* prime implicants. Only $\overline{A}\,\overline{B}$ is a prime implicant in that K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

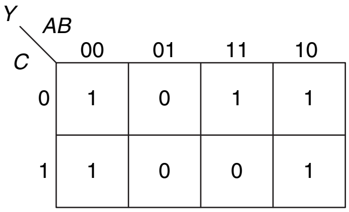
Example 2.9 MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP

Suppose we have the function $Y = F(A, B, C)$ with the K-map shown in Figure 2.45. Minimize the equation using the K-map.

Solution: Circle the 1's in the K-map using as few circles as possible, as shown in Figure 2.46. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two (2×1 and 2×2). We form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.

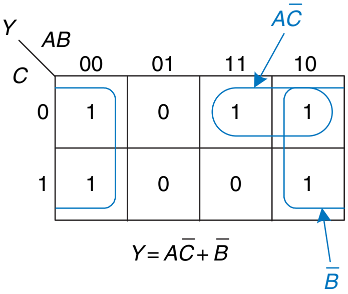
For example, in the 2×1 circle, the true and complementary forms of B are included in the circle, so we *do not* include B in the prime implicant. However, only the true form of A (A) and complementary form of C ($\overline{C}$) are in this circle, so we include these variables in the prime implicant $A\overline{C}$. Similarly, the 2×2 circle covers all squares where $B = 0$, so the prime implicant is $\overline{B}$.

Notice how the top-right square (minterm) is covered twice to make the prime implicant circles as large as possible. As we saw with Boolean algebra techniques, this is equivalent to sharing a minterm to reduce the size of the implicant. Also notice how the circle covering four squares wraps around the sides of the K-map.



C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Figure 2.45 K-map for Example 2.9



$C \backslash AB$	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Figure 2.46 **Solution for Example 2.9**

Example 2.10 SEVEN-SEGMENT DISPLAY DECODER

A *seven-segment display decoder* takes a 4-bit data input $D_{3:0}$ and produces seven outputs to control light-emitting diodes to display a digit from 0 to 9. The seven outputs are often called segments *a* through *g*, or S_a–S_g, as defined in Figure 2.47. The digits are shown in Figure 2.48. Write a truth table for the outputs, and use K-maps to find Boolean equations for outputs S_a and S_b. Assume that illegal input values (10–15) produce a blank readout.

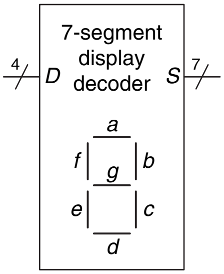


Figure 2.47 **Seven-segment display decoder icon**

Solution: The truth table is given in Table 2.6. For example, an input of 0000 should turn on all segments except S_g.

Each of the seven outputs is an independent function of four variables. The K-maps for outputs S_a and S_b are shown in Figure 2.49. Remember that adjacent squares may differ in only a single variable, so we label the rows and columns in Gray code order: 00, 01, 11, 10. Be careful to also remember this ordering when *entering* the output values into the squares.

Next, circle the prime implicants. Use the fewest number of circles necessary to cover all the 1's. A circle can wrap around the edges (vertical *and* horizontal), and a 1 may be circled more than once. Figure 2.50 shows the prime implicants and the simplified Boolean equations.

Note that the minimal set of prime implicants is not unique. For example, the 0000 entry in the S_a K-map was circled along with the 1000 entry to produce the $\overline{D}_2\overline{D}_1\overline{D}_0$ minterm. The circle could have included the 0010 entry instead, producing a $\overline{D}_3\overline{D}_2\overline{D}_0$ minterm, as shown with dashed lines in Figure 2.51.

Figure 2.52 (see page 82) illustrates a common error in which a nonprime implicant was chosen to cover the 1 in the upper left corner. This minterm, $\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$, gives a sum-of-products equation that is *not* minimal. The minterm could have been combined with either of the adjacent ones to form a larger circle, as was done in the previous two figures.

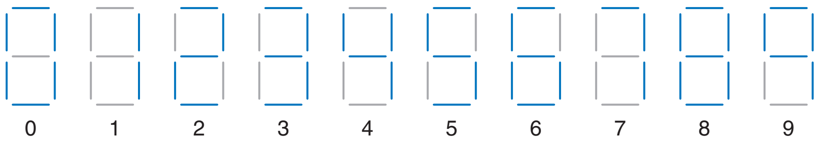
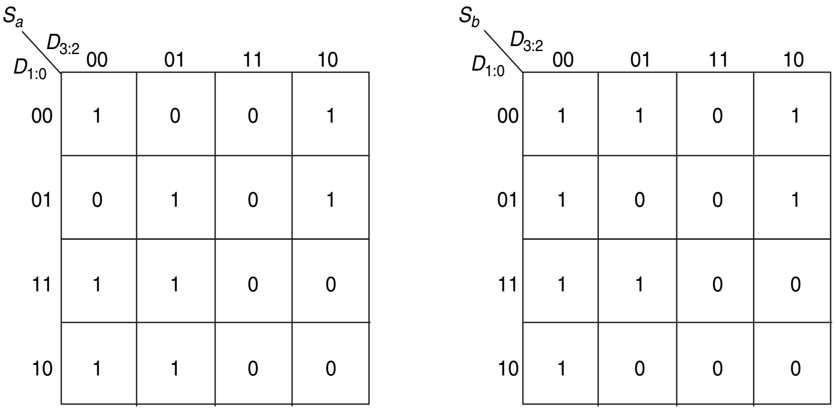


Figure 2.48 Seven-segment display digits

Table 2.6 Seven-segment display decoder truth table

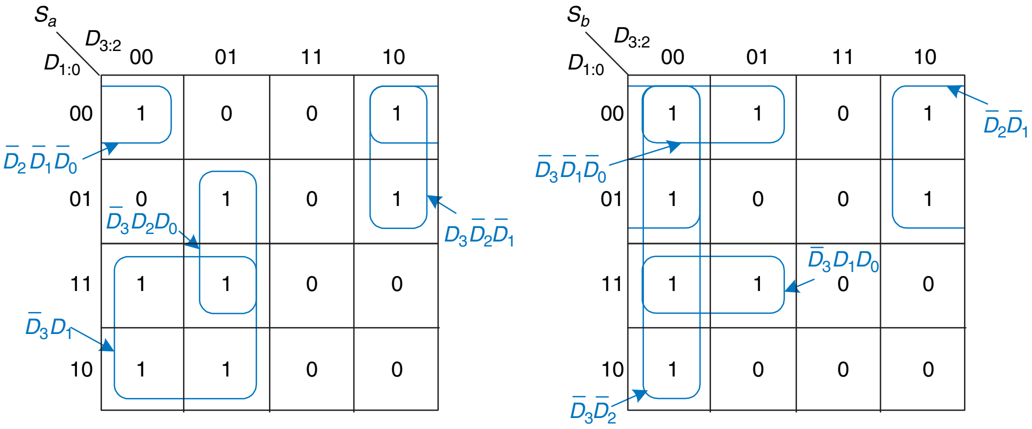
$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0



S_a: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

S_b: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

Figure 2.49 Karnaugh maps for S_a and S_b


S_a
$D_{3:2}$
$D_{1:0}$
$\overline{D_2}\,\overline{D_1}\,\overline{D_0}$
$\overline{D_3}D_2D_0$
$D_3\overline{D_2}\,\overline{D_1}$
$\overline{D_3}D_1$
S_b
$D_{3:2}$
$D_{1:0}$
$\overline{D_2}\,\overline{D_1}$
$\overline{D_3}\,\overline{D_1}\,\overline{D_0}$
$\overline{D_3}D_1D_0$
$\overline{D_3}\,\overline{D_2}$

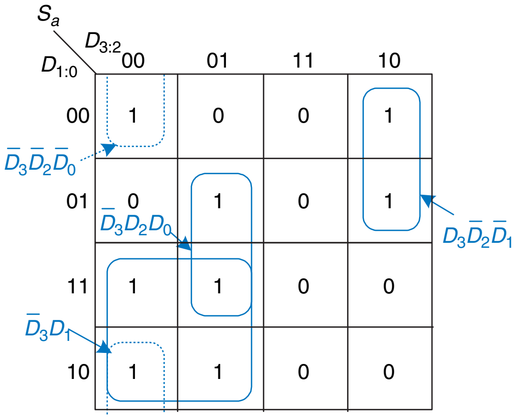
S_a	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

S_b	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

$$S_a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_2}\,\overline{D_1}\,\overline{D_0}$$

$$S_b = \overline{D_3}\,\overline{D_2} + \overline{D_2}\,\overline{D_1} + \overline{D_3}D_1D_0 + \overline{D_3}\,\overline{D_1}\,\overline{D_0}$$

Figure 2.50 **K-map solution for Example 2.10**


S_a
$D_{3:2}$
$D_{1:0}$
$\overline{D_3}\,\overline{D_2}\,\overline{D_0}$
$\overline{D_3}D_2D_0$
$D_3\overline{D_2}\,\overline{D_1}$
$\overline{D_3}D_1$

S_a	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

$$S_a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_3}\,\overline{D_2}\,\overline{D_0}$$

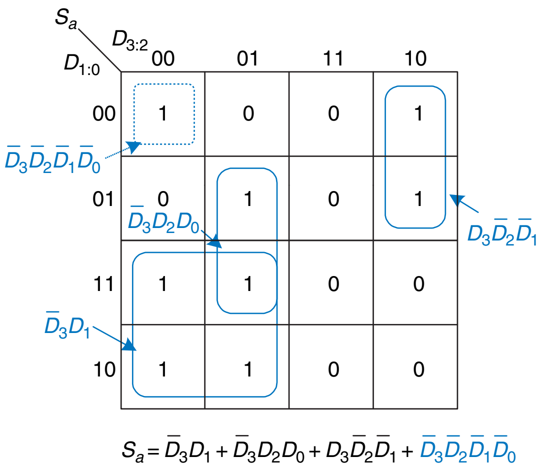
Figure 2.51 **Alternative K-map for S_a showing different set of prime implicants**

2.7.3 Don't Cares

Recall that "don't care" entries for truth table inputs were introduced in Section 2.4 to reduce the number of rows in the table when some variables do not affect the output. They are indicated by the symbol X, which means that the entry can be either 0 or 1.

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never

Figure 2.52 **Alternative K-map for S_a showing incorrect nonprime implicant**



S_a
$D_{3:2}$
$D_{1:0}$
00
01
11
10
00
01
11
10
1
0
0
1
0
1
0
1
1
1
0
0
1
1
0
0
$\overline{D_3}\,\overline{D_2}\,\overline{D_1}\,\overline{D_0}$
$\overline{D_3}D_2D_0$
$D_3\overline{D_2}\,\overline{D_1}$
$\overline{D_3}D_1$
$S_a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_3}\,\overline{D_2}\,\overline{D_1}\,\overline{D_0}$

happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.

In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

Example 2.11 SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES

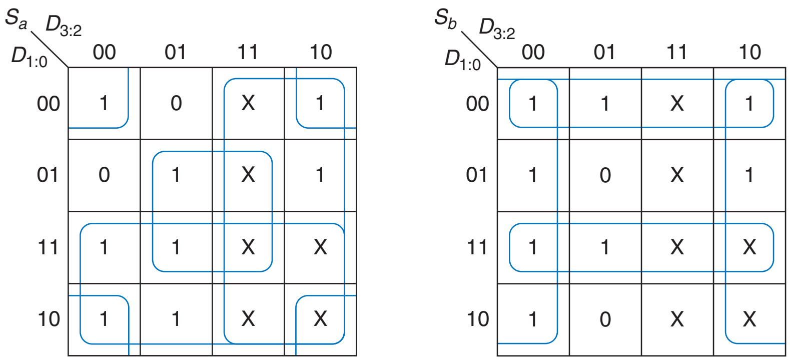
Repeat Example 2.10 if we don't care about the output values for illegal input values of 10 to 15.

Solution: The K-map is shown in Figure 2.53 with X entries representing don't care. Because don't cares can be 0 or 1, we circle a don't care if it allows us to cover the 1's with fewer or bigger circles. Circled don't cares are treated as 1's, whereas uncircled don't cares are 0's. Observe how a 2 × 2 square wrapping around all four corners is circled for segment S_a. Use of don't cares simplifies the logic substantially.

2.7.4 The Big Picture

Boolean algebra and Karnaugh maps are two methods of logic simplification. Ultimately, the goal is to find a low-cost method of implementing a particular logic function.

In modern engineering practice, computer programs called *logic synthesizers* produce simplified circuits from a description of the logic function, as we will see in Chapter 4. For large problems, logic synthesizers are much more efficient than humans. For small problems, a



$$S_a = D_3 + D_2D_0 + \overline{D}_2\overline{D}_0 + D_1$$

$$S_b = \overline{D}_2 + D_1D_0 + \overline{D}_1\overline{D}_0$$

Figure 2.53 K-map solution with don't cares

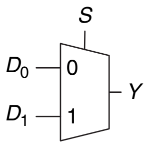
human with a bit of experience can find a good solution by inspection. Neither of the authors has ever used a Karnaugh map in real life to solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 2:1 multiplexer symbol and truth table

2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs D_0 and D_1, a select input S, and one output Y. The multiplexer chooses between the two data inputs based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$. S is also called a *control signal* because it controls what the multiplexer does.

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived

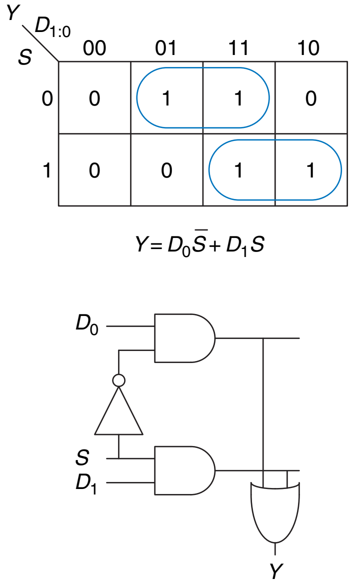


Figure 2.55 **2:1 multiplexer implementation using two-level logic**

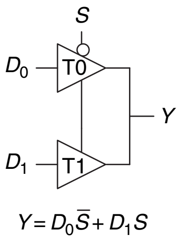


Figure 2.56 **Multiplexer using tristate buffers**

with a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

Alternatively, multiplexers can be built from tristate buffers as shown in Figure 2.56. The tristate enables are arranged such that, at all times, exactly one tristate buffer is active. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y. When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y.

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N:1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

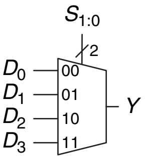


Figure 2.57 **4:1 multiplexer**

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

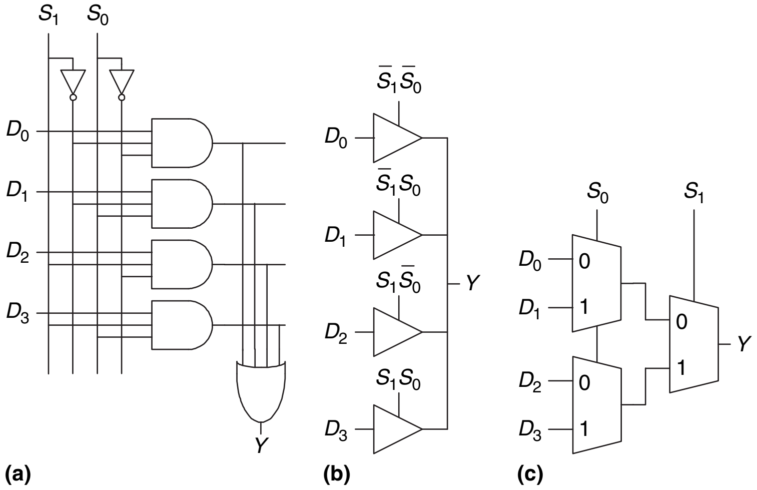


Figure 2.58 **4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical**

AND gate. The inputs, A and B, serve as select lines. The multiplexer data inputs are connected to 0 or 1 according to the corresponding row of the truth table. In general, a 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1}-input multiplexer to perform any N-input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table, and then combine pairs of rows to eliminate the rightmost input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of B. When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

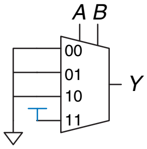


Figure 2.59 **4:1 multiplexer implementation of two-input AND function**

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution: Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table where each row in the truth table corresponds to a multiplexer input.

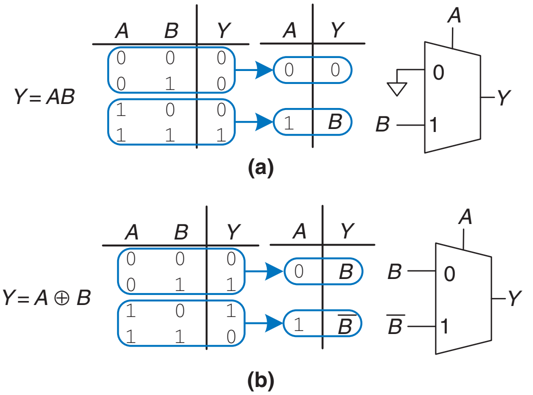
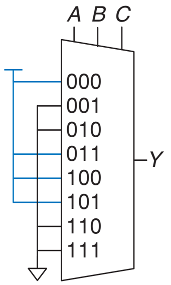


Figure 2.60 Multiplexer logic using variable inputs

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$

(a)



(b)

Figure 2.61 Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation

---

Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidently powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution: Alyssa reduces her truth table to four rows by letting the output depend on *C*. (She could also have chosen to rearrange the columns of the truth table to let the output depend on *A* or *B*.) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.

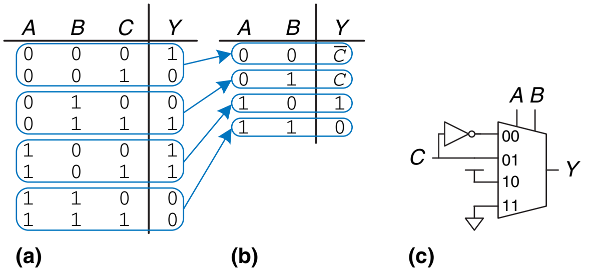
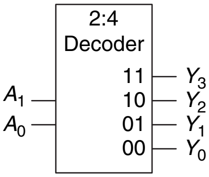

A B C Y
A B Y
$\overline{C}$
C
A B
C
00
01
10
11
Y
(a)
(b)
(c)

Figure 2.62 Alyssa's new circuit

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND, OR, and NOT gates.

Solution: Figure 2.64 shows an implementation for the 2:4 decoder using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an $N{:}2^N$ decoder can be constructed from 2^N N-input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A_1}\,\overline{A_0}$. This fact will be handy when using decoders with other digital building blocks.


2:4
Decoder
A_1
A_0
11
10
01
00
Y_3
Y_2
Y_1
Y_0

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 2:4 decoder

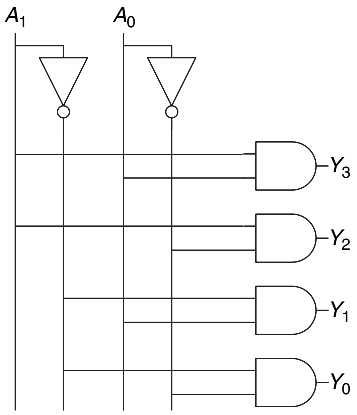

A_1
A_0
Y_3
Y_2
Y_1
Y_0

Figure 2.64 2:4 decoder implementation

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all the minterms in the function. In Figure 2.65, $Y = \overline{A}\,\overline{B} + AB = \overline{A \oplus B}$.

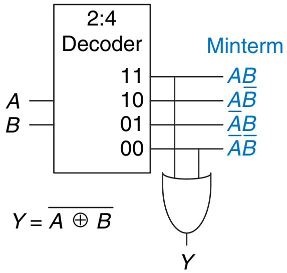

2:4
Decoder
A
B
11
10
01
00
Minterm
AB
$A\overline{B}$
$\overline{A}B$
$\overline{A}\,\overline{B}$
$Y = \overline{A \oplus B}$
Y

Figure 2.65 Logic function using decoder

When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. An N-input function with M 1's in the truth table can be built with an $N{:}2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table. This concept will be applied to the building of Read Only Memories (ROMs) in Section 5.5.6.

2.9 TIMING

In previous sections, we have been concerned primarily with whether the circuit works—ideally, using the fewest gates. However, as any seasoned circuit designer will attest, one of the most challenging issues in circuit design is *timing:* making a circuit run fast.

An output takes time to change in response to an input change. Figure 2.66 shows the *delay* between an input change and the subsequent output change for a buffer. The figure is called a *timing diagram;* it portrays the *transient response* of the buffer circuit when an input changes. The transition from LOW to HIGH is called the *rising edge*. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the *falling edge*. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A. We measure delay from the *50% point* of the input signal, A, to the 50% point of the output signal, Y. The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions.

When designers speak of calculating the *delay* of a circuit, they generally are referring to the worst-case value (the propagation delay), unless it is clear otherwise from the context.

2.9.1 Propagation and Contamination Delay

Combinational logic is characterized by its *propagation delay* and *contamination delay*. The propagation delay t_{pd} is the maximum time from when an input changes until the output or outputs reach their final value. The contamination delay t_{cd} is the minimum time from when an input changes until any output starts to change its value.

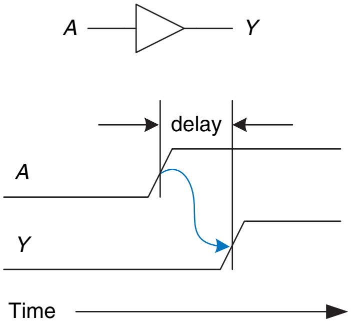


Figure 2.66 Circuit delay

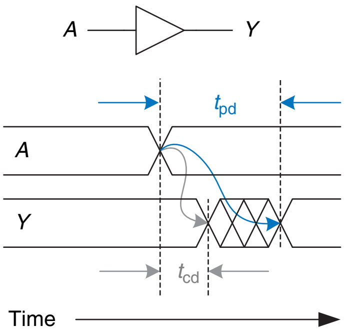


Figure 2.67 **Propagation and contamination delay**

Figure 2.67 illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps = 10^{-12} seconds) to nanoseconds (1 ns = 10^{-9} seconds). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Calculating t_{pd} and t_{cd} requires delving into the lower levels of abstraction beyond the scope of this book. However, manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.68 shows a four-input logic circuit. The *critical path*, shown in blue, is the path from input A or B to output Y. It is the longest, and

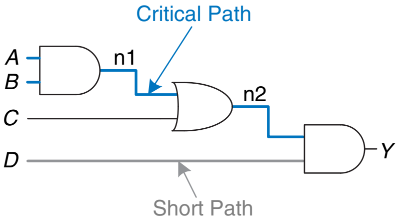


Figure 2.68 **Short path and critical path**

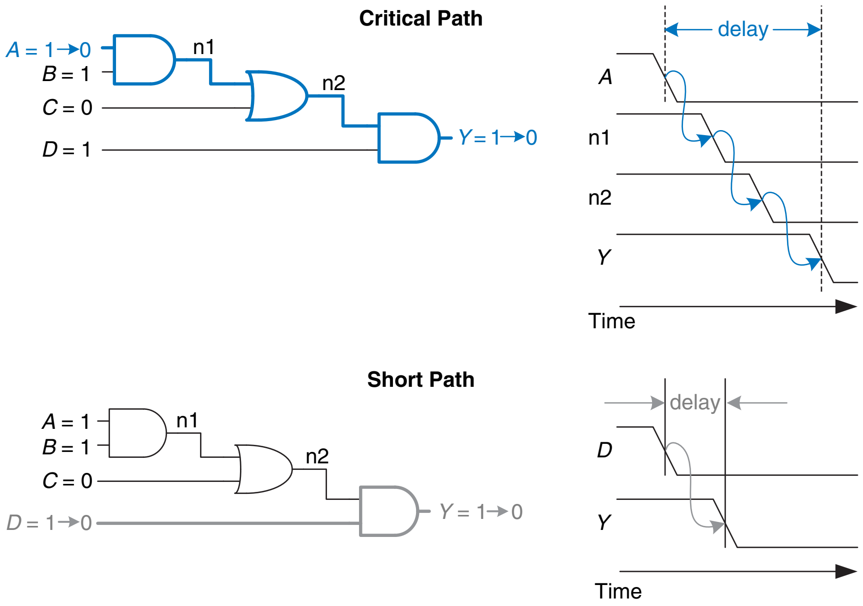


Figure 2.69 **Critical and short path waveforms**

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in wires is covered in Appendix A.

therefore the slowest, path, because the input travels through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The *short path* through the circuit, shown in gray, is from input D to output Y. This is the shortest, and therefore the fastest, path through the circuit, because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure 2.69 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\text{AND}} + t_{pd\_\text{OR}} \tag{2.8}$$

$$t_{cd} = t_{cd\_\text{AND}} \tag{2.9}$$

Example 2.15 FINDING DELAYS

Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure 2.70. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.

Solution: Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue in Figure 2.71, is from input *A* or *B* through three gates to the output *Y*. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray in Figure 2.72, is from input *C, D*, or *E* through two gates to the output *Y*. There are only two gates in the shortest path, so t_{cd} is 120 ps.

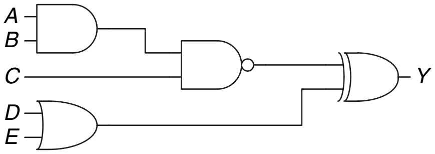


Figure 2.70 Ben's circuit

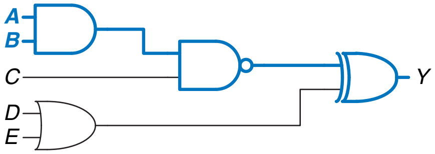


Figure 2.71 Ben's critical path

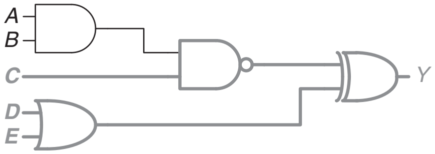


Figure 2.72 Ben's shortest path

Example 2.16 MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL

Compare the worst-case timing of the three four-input multiplexer designs shown in Figure 2.58 in Section 2.8.1. Table 2.7 lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Solution: One of the critical paths for each of the three design options is highlighted in blue in Figures 2.73 and 2.74. t_{pd_sy} indicates the propagation delay from input *S* to output *Y*; t_{pd_dy} indicates the propagation delay from input *D* to output *Y*; t_{pd} is the worst of the two: $\max(t_{pd\_sy}, t_{pd\_dy})$.

For both the two-level logic and tristate implementations in Figure 2.73, the critical path is from one of the control signals *S* to the output *Y*: $t_{pd} = t_{pd\_sy}$. These circuits are *control critical*, because the critical path is from the control signals to the output. Any additional delay in the control signals will add directly to the worst-case delay. The delay from *D* to *Y* in Figure 2.73(b) is only 50 ps, compared with the delay from *S* to *Y* of 125 ps.

Figure 2.74 shows the hierarchical implementation of the 4:1 multiplexer using two stages of 2:1 multiplexers. The critical path is from any of the D inputs to the output. This circuit is *data critical*, because the critical path is from the data input to the output: $t_{pd} = t_{pd\_dy}$.

If data inputs arrive well before the control inputs, we would prefer the design with the shortest control-to-output delay (the hierarchical design in Figure 2.74). Similarly, if the control inputs arrive well before the data inputs, we would prefer the design with the shortest data-to-output delay (the tristate design in Figure 2.73(b)).

The best choice depends not only on the critical path through the circuit and the input arrival times, but also on the power, cost, and availability of parts.

Table 2.7 **Timing specifications for multiplexer circuit elements**

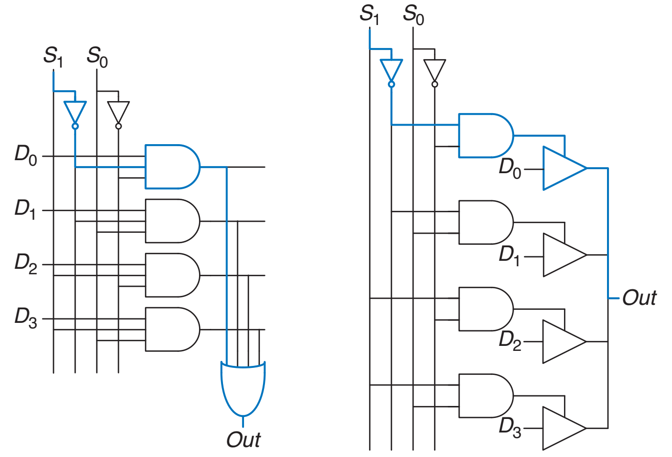
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

2.9.2 Glitches

So far we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called *glitches* or *hazards*. Although glitches usually don't cause problems, it is important to realize that they exist and recognize them when looking at timing diagrams. Figure 2.75 shows a circuit with a glitch and the Karnaugh map of the circuit.

Hazards have another meaning related to microarchitecture in Chapter 7, so we will stick with the term *glitches* for multiple output transitions to avoid confusion.

The Boolean equation is correctly minimized, but let's look at what happens when $A = 0$, $C = 1$, and B transitions from 1 to 0. Figure 2.76 (see page 94) illustrates this scenario. The short path (shown in gray) goes through two gates, the AND and OR gates. The critical path (shown in blue) goes through an inverter and two gates, the AND and OR gates.



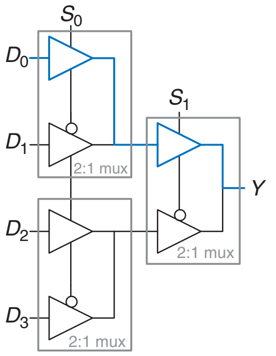
(a)

$$t_{pd\_sy} = t_{pd\_\text{INV}} + t_{pd\_\text{AND3}} + t_{pd\_\text{OR4}}$$
$$= 30\text{ ps} + 80\text{ ps} + 90\text{ ps}$$
$$= \mathbf{200\text{ ps}}$$
$$t_{pd\_dy} = t_{pd\_\text{AND3}} + t_{pd\_\text{OR4}}$$
$$= \mathbf{170\text{ ps}}$$

(b)

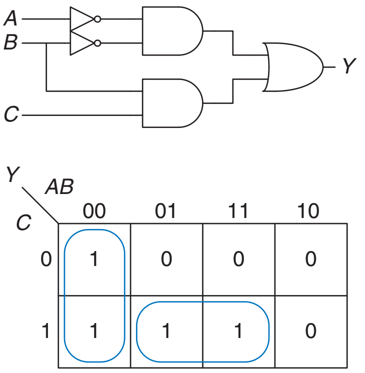
$$t_{pd\_sy} = t_{pd\_\text{INV}} + t_{pd\_\text{AND2}} + t_{pd\_\text{TRI}\_sy}$$
$$= 30\text{ ps} + 60\text{ ps} + 35\text{ ps}$$
$$= \mathbf{125\text{ ps}}$$
$$t_{pd\_dy} = t_{pd\_\text{TRI}\_ay}$$
$$= \mathbf{50\text{ ps}}$$

Figure 2.73 **4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate**



$$t_{pd\_s0y} = t_{pd\_\text{TRI}\_sy} + t_{pd\_\text{TRI}\_ay} = \mathbf{85\text{ ps}}$$
$$t_{pd\_dy} = 2\ t_{pd\_\text{TRI}\_ay} = \mathbf{100\text{ ps}}$$

Figure 2.74 **4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers**



C \ AB	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$$Y = \overline{A}\,\overline{B} + BC$$

Figure 2.75 **Circuit with a glitch**

As *B* transitions from 1 to 0, n2 (on the short path) falls before n1 (on the critical path) can rise. Until n1 rises, the two inputs to the OR gate are 0, and the output *Y* drops to 0. When n1 eventually rises, *Y* returns to 1. As shown in the timing diagram of Figure 2.76, *Y* starts at 1 and ends at 1 but momentarily glitches to 0.

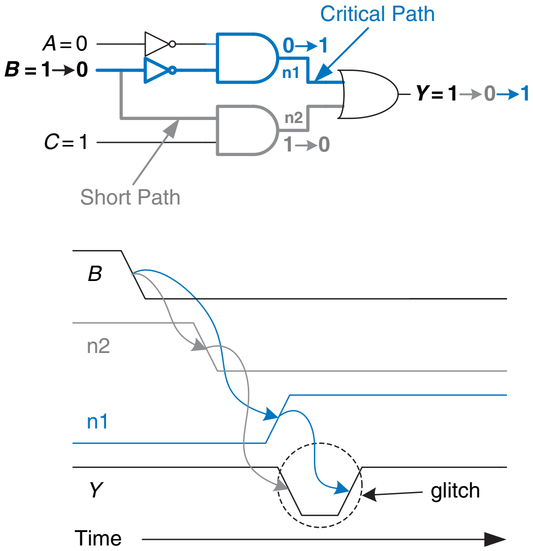

Critical Path
A = 0
B = 1→0
C = 1
0→1
n1
n2
1→0
Y = 1→0→1
Short Path
B
n2
n1
Y
glitch
Time

Figure 2.76 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are not a problem, because the output eventually settles to the right answer.

If we choose to, we can avoid this glitch by adding another gate to the implementation. This is easiest to understand in terms of the K-map. Figure 2.77 shows how an input transition on B from $ABC = 011$ to $ABC = 001$ moves from one prime implicant circle to another. The transition across the boundary of two prime implicants in the K-map indicates a possible glitch.

As we saw from the timing diagram in Figure 2.76, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.78. You might recognize this as the consensus theorem, where the added term, $\overline{A}C$, is the consensus or redundant term.

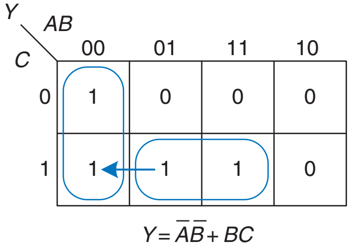

Y
AB
C
00
01
11
10
0
1
1
0
0
0
1
1
1
1
0
$Y = \overline{A}\overline{B} + BC$

Figure 2.77 Input change crosses implicant boundary

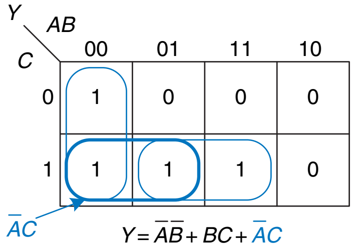


Figure 2.78 **K-map without glitch**

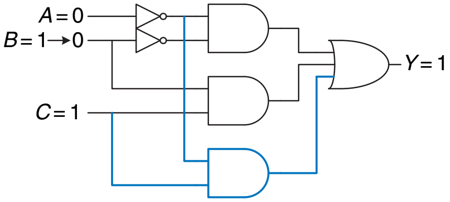


Figure 2.79 **Circuit without glitch**

Figure 2.79 shows the glitch-proof circuit. The added AND gate is highlighted in blue. Now a transition on B when $A = 0$ and $C = 1$ does not cause a glitch on the output, because the blue AND gate outputs 1 throughout the transition.

In general, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding redundant implicants to the K-map to cover these boundaries. This of course comes at the cost of extra hardware.

However, simultaneous transitions on multiple inputs can also cause glitches. These glitches cannot be fixed by adding hardware. Because the vast majority of interesting systems have simultaneous (or near-simultaneous) transitions on multiple inputs, glitches are a fact of life in most circuits. Although we have shown how to eliminate one kind of glitch, the point of discussing glitches is not to eliminate them but to be aware that they exist. This is especially important when looking at timing diagrams on a simulator or oscilloscope.

2.10 SUMMARY

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module. This chapter has focused on combinational circuits, circuits whose outputs depend only on the current values of the inputs.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products or product-of-sums form. In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equations can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$. Karnaugh maps are a visual tool for minimizing functions of up to four variables. With practice, designers can usually simplify functions of a few variables by inspection. Computer-aided design tools are used for more complicated functions; such methods and tools are discussed in Chapter 4.

Logic gates are connected to create combinational circuits that perform the desired function. Any function in sum-of-products form can be built using two-level logic: NOT gates form the complements of the inputs, AND gates form the products, and OR gates form the sum. Depending on the function and the building blocks available, multilevel logic implementations with various types of gates may be more efficient. For example, CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates. When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits such as multiplexers, decoders, and priority circuits. A multiplexer chooses one of the data inputs based on the select input. A decoder sets one of the outputs HIGH according to the inputs. A priority circuit produces an output indicating the highest priority input. These circuits are all examples of combinational building blocks. Chapter 5 will introduce more building blocks, including other arithmetic circuits. These building blocks will be used extensively to build a microprocessor in Chapter 7.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit. These indicate the longest and shortest times between an input change and the consequent output change. Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path. There are many different ways to implement complicated combinational circuits; these ways offer trade-offs between speed and cost.

The next chapter will move to sequential circuits, whose outputs depend on current as well as previous values of the inputs. In other words, sequential circuits have *memory* of the past.

Exercises

Exercise 2.1 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.80.

(a)

A	*B*	*Y*
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.80 **Truth tables for Exercises 2.1 and 2.3**

Exercise 2.2 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.81.

(a)

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 2.81 **Truth tables for Exercises 2.2 and 2.4**

Exercise 2.3 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.80.

Exercise 2.4 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.81.

Exercise 2.5 Minimize each of the Boolean equations from Exercise 2.1.

Exercise 2.6 Minimize each of the Boolean equations from Exercise 2.2.

Exercise 2.7 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.5. Reasonably simple means that you are not wasteful of gates, but you don't waste vast amounts of time checking every possible implementation of the circuit either.

Exercise 2.8 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.6.

Exercise 2.9 Repeat Exercise 2.7 using only NOT gates and AND and OR gates.

Exercise 2.10 Repeat Exercise 2.8 using only NOT gates and AND and OR gates.

Exercise 2.11 Repeat Exercise 2.7 using only NOT gates and NAND and NOR gates.

Exercise 2.12 Repeat Exercise 2.8 using only NOT gates and NAND and NOR gates.

Exercise 2.13 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + (\overline{A + \overline{C}})$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Exercise 2.14 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = \overline{A}BC + \overline{A}B\overline{C}$

(b) $Y = \overline{ABC} + A\overline{B}$

(c) $Y = ABC\overline{D} + A\overline{BCD} + (\overline{A + B + C + D})$

Exercise 2.15 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.13.

Exercise 2.16 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.14.

Exercise 2.17 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D$
$+ \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Exercise 2.18 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$

(b) $Y = (\overline{A + B + C})D + AD + B$

(c) $Y = ABCD + \overline{A}B\overline{C}D + \overline{(\overline{B} + D)}E$

Exercise 2.19 Give an example of a truth table requiring between 3 billion and 5 billion rows that can be constructed using fewer than 40 (but at least 1) two-input gates.

Exercise 2.20 Give an example of a circuit with a cyclic path that is nevertheless combinational.

Exercise 2.21 Alyssa P. Hacker says that any Boolean function can be written in minimal sum-of-products form as the sum of all of the prime implicants of the function. Ben Bitdiddle says that there are some functions whose minimal equation does not involve all of the prime implicants. Explain why Alyssa is right or provide a counterexample demonstrating Ben's point.

Exercise 2.22 Prove that the following theorems are true using perfect induction. You need not prove their duals.

(a) The idempotency theorem (T3)

(b) The distributivity theorem (T8)

(c) The combining theorem (T10)

Exercise 2.23 Prove De Morgan's Theorem (T12) for three variables, B_2, B_1, B_0, using perfect induction.

Exercise 2.24 Write Boolean equations for the circuit in Figure 2.82. You need not minimize the equations.

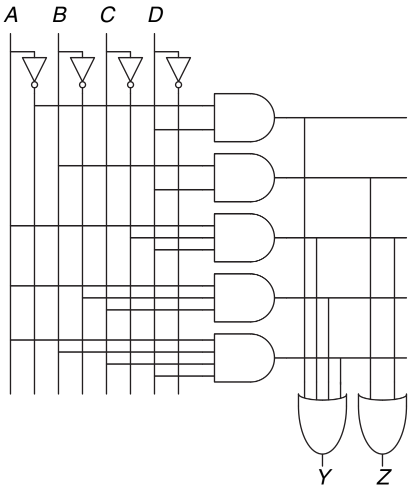


Figure 2.82 **Circuit schematic**

Exercise 2.25 Minimize the Boolean equations from Exercise 2.24 and sketch an improved circuit with the same function.

Exercise 2.26 Using De Morgan equivalent gates and bubble pushing methods, redraw the circuit in Figure 2.83 so that you can find the Boolean equation by inspection. Write the Boolean equation.

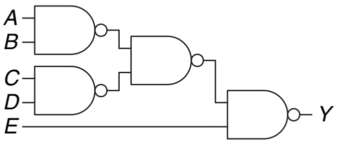


Figure 2.83 **Circuit schematic**

Exercise 2.27 Repeat Exercise 2.26 for the circuit in Figure 2.84.

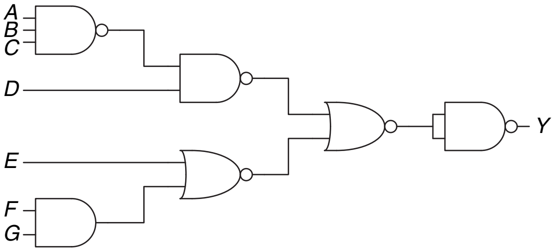


Figure 2.84 **Circuit schematic**

Exercise 2.28 Find a minimal Boolean equation for the function in Figure 2.85. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.85 **Truth table for Exercise 2.28**

Exercise 2.29 Sketch a circuit for the function from Exercise 2.28.

Exercise 2.30 Does your circuit from Exercise 2.29 have any potential glitches when one of the inputs changes? If not, explain why not. If so, show how to modify the circuit to eliminate the glitches.

Exercise 2.31 Find a minimal Boolean equation for the function in Figure 2.86. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.86 **Truth table for Exercise 2.31**

Exercise 2.32 Sketch a circuit for the function from Exercise 2.31.

Exercise 2.33 Ben Bitdiddle will enjoy his picnic on sunny days that have no ants. He will also enjoy his picnic any day he sees a hummingbird, as well as on days where there are ants and ladybugs. Write a Boolean equation for his enjoyment (E) in terms of sun (S), ants (A), hummingbirds (H), and ladybugs (L).

Exercise 2.34 Complete the design of the seven-segment decoder segments S_c through S_g (see Example 2.10):

(a) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 must produce blank (0) outputs.

(b) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 are don't cares.

(c) Sketch a reasonably simple gate-level implementation of part (b). Multiple outputs can share gates where appropriate.

Exercise 2.35 A circuit has four inputs and two outputs. The inputs $A_{3:0}$ represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.

Exercise 2.36 A *priority encoder* has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE, or 0 if none of the inputs are TRUE. It also produces an output $NONE$ that is TRUE if none of the inputs are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2.0}$ and *NONE*. For example, if the input is 00100000, the output *Y* should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.37 Design a modified priority encoder (see Exercise 2.36) that receives an 8-bit input, $A_{7:0}$, and produces two 3-bit outputs, $Y_{2:0}$ and $Z_{2:0}$ *Y* indicates the most significant bit of the input that is TRUE. *Z* indicates the second most significant bit of the input that is TRUE. *Y* should be 0 if none of the inputs are TRUE. *Z* should be 0 if no more than one of the inputs is TRUE. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.38 An *M*-bit *thermometer code* for the number *k* consists of *k* 1's in the least significant bit positions and *M* – *k* 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has *N* inputs and 2^N–1 outputs. It produces a 2^N–1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.39 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.87.

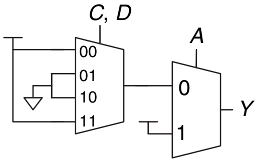


Figure 2.87 Multiplexer circuit

Exercise 2.40 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.88.

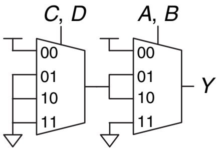


Figure 2.88 Multiplexer circuit

Exercise 2.41 Implement the function from Figure 2.80(b) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and one inverter

(c) a 2:1 multiplexer and two other logic gates

Exercise 2.42 Implement the function from Exercise 2.17(a) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and no other gates

(c) a 2:1 multiplexer, one OR gate, and an inverter

Exercise 2.43 Determine the propagation delay and contamination delay of the circuit in Figure 2.83. Use the gate delays given in Table 2.8.

Exercise 2.44 Determine the propagation delay and contamination delay of the circuit in Figure 2.84. Use the gate delays given in Table 2.8.

Table 2.8 **Gate delays for Exercises 2.43–2.47**

Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

Exercise 2.45 Sketch a schematic for a fast 3:8 decoder. Suppose gate delays are given in Table 2.8 (and only the gates in that table are available). Design your decoder to have the shortest possible critical path, and indicate what that path is. What are its propagation delay and contamination delay?

Exercise 2.46 Design an 8:1 multiplexer with the shortest possible delay from the data inputs to the output. You may use any of the gates from Table 2.7 on page 92. Sketch a schematic. Using the gate delays from the table, determine this delay.

Exercise 2.47 Redesign the circuit from Exercise 2.35 to be as fast as possible. Use only the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.48 Redesign the priority encoder from Exercise 2.36 to be as fast as possible. You may use any of the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 2.1 Sketch a schematic for the two-input XOR function using only NAND gates. How few can you use?

Question 2.2 Design a circuit that will tell whether a given month has 31 days in it. The month is specified by a 4-bit input $A_{3:0}$. For example, if the inputs are 0001, the month is January, and if the inputs are 1100, the month is December. The circuit output Y should be HIGH only when the month specified by the inputs has 31 days in it. Write the simplified equation, and draw the circuit diagram using a minimum number of gates. (Hint: Remember to take advantage of don't cares.)

Question 2.3 What is a tristate buffer? How and why is it used?

Question 2.4 A gate or set of gates is universal if it can be used to construct any Boolean function. For example, the set {AND, OR, NOT} is universal.

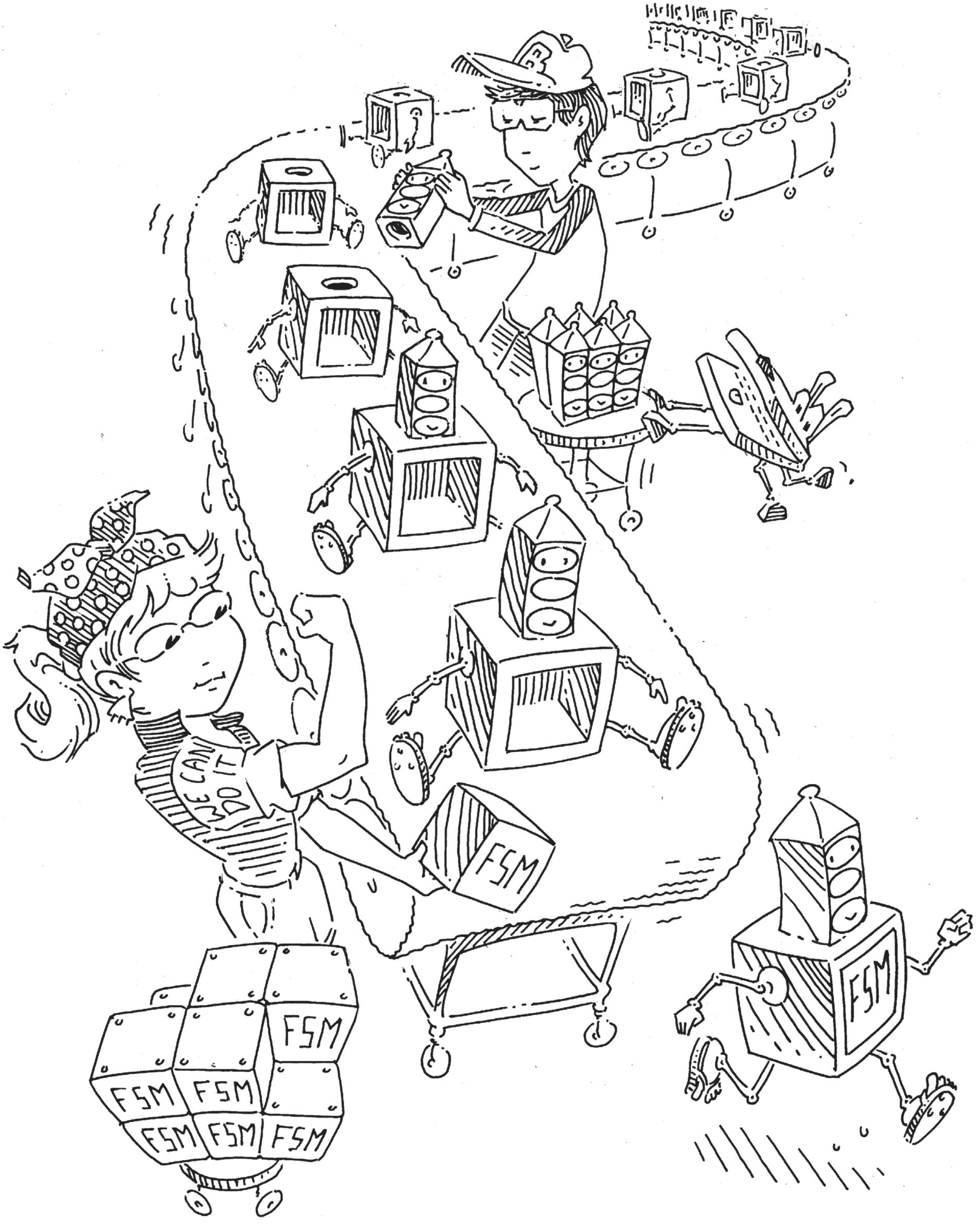
(a) Is an AND gate by itself universal? Why or why not?

(b) Is the set {OR, NOT} universal? Why or why not?

(c) Is a NAND gate by itself universal? Why or why not?

Question 2.5 Explain why a circuit's contamination delay might be less than (instead of equal to) its propagation delay.

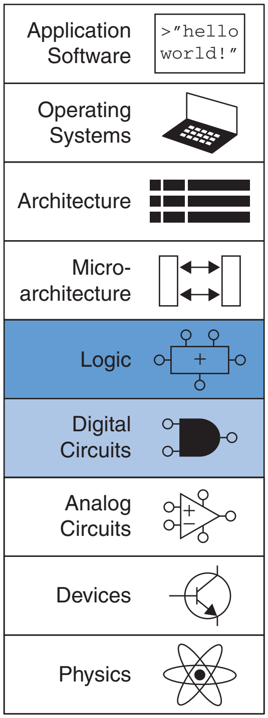
This page intentionally left blank


WE CAN DO IT
FSM
FSM
FSM
FSM
FSM
FSM
FSM
FSM

3

Sequential Logic Design

3.1 **Introduction**
3.2 **Latches and Flip-Flops**
3.3 **Synchronous Logic Design**
3.4 **Finite State Machines**
3.5 **Timing of Sequential Logic**
3.6 **Parallelism**
3.7 **Summary**
Exercises
Interview Questions



3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase speed.

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs, Q and $\overline{Q}$.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00003-3
© 2013 Elsevier, Inc. All rights reserved.

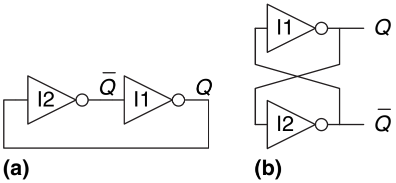


Figure 3.1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on $\overline{Q}$, and $\overline{Q}$ depends on Q.

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- *Case I:* $Q = 0$
  As shown in Figure 3.2(a), I2 receives a FALSE input, Q, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on Q. This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.

- *Case II:* $Q = 1$
  As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input and produces a TRUE output on Q. This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is known, $\overline{Q}$ is also known. On the other hand, $\overline{Q}$ is also an acceptable choice for the state variable.

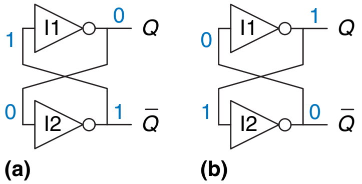


Figure 3.2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.

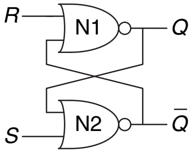


Figure 3.3 **SR latch schematic**

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S.

- *Case I:* $R = 1, S = 0$
  N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.

- *Case II:* $R = 0, S = 1$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.

- *Case III:* $R = 1, S = 1$
  N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and $\overline{Q}$ are both FALSE.

- *Case IV:* $R = 0, S = 0$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these subcases.

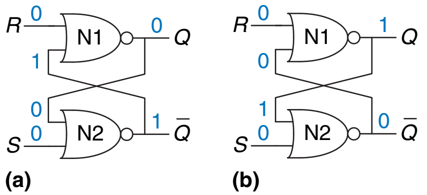


Figure 3.4 **Bistable states of SR latch**

- *Case IVa:* $Q = 0$
  Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$, as shown in Figure 3.4(a). Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE, just as we had assumed.
- *Case IVb:* $Q = 1$
  Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$, as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE, just as we had assumed.

Putting this all together, suppose Q has some known prior value, which we will call Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 **SR latch truth table**

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and $\overline{Q}$, are normally complementary. When R is asserted, Q is reset to 0 and $\overline{Q}$ does the opposite. When S is asserted, Q is set to 1 and $\overline{Q}$ does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev}. Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

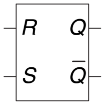


Figure 3.6 **SR latch symbol**

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be accounted for by the single state variable Q. No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

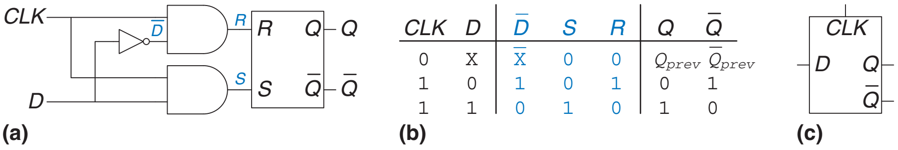
The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes $\overline{D}$, S, and R. If $CLK = 0$, both S and R are FALSE, regardless of the value of D. If $CLK = 1$, one AND gate will produce TRUE and the other FALSE, depending on the value of D. Given S and R, Q and $\overline{Q}$ are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev}. When $CLK = 1$, $Q = D$. In all cases, $\overline{Q}$ is the complement of Q, as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.



CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

3.2.3 D Flip-Flop

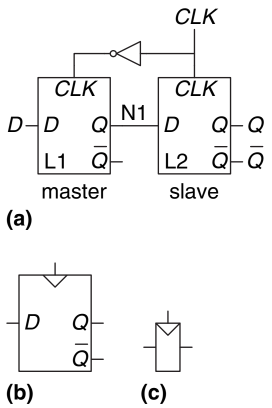


Figure 3.8 **D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol**

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the $\overline{Q}$ output is not needed, the symbol is often condensed as in Figure 3.8(c).

When $CLK = 0$, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D. Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

In other words, *a D flip-flop copies D to Q on the rising edge of the clock*, *and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The $\overline{Q}$ output is often omitted when it is not needed.

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term flip-flop or latch by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution: A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation using transmission gates.

3.2.4 Register

An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time. Registers are the key building block of most sequential circuits. Figure 3.9

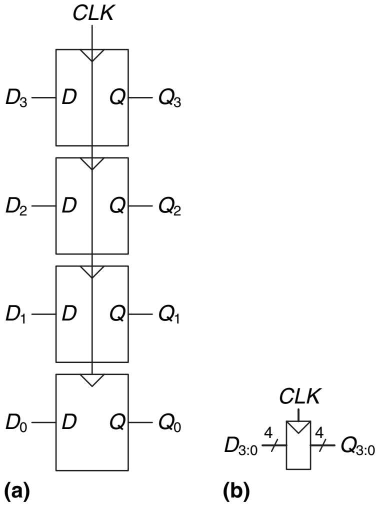


Figure 3.9 **A 4-bit register: (a) schematic and (b) symbol**

shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer chooses whether to pass the value at *D*, if *EN* is TRUE, or to recycle the old state from *Q*, if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*. If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is

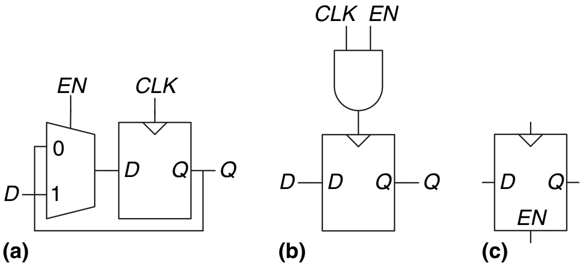


Figure 3.10 **Enabled flip-flop: (a, b) schematics, (c) symbol**

FALSE, the *CLK* input is also FALSE and the flip-flop retains its old value. Notice that *EN* must not change while $CLK = 1$, lest the flip-flop see a clock *glitch* (switch at an incorrect time). Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors, as we will see in Section 3.5.3, so do it only if you are sure you know what you are doing. The symbol for an enabled flip-flop is given in Figure 3.10(c).

3.2.6 Resettable Flip-Flop

A *resettable flip-flop* adds another input called *RESET*. When *RESET* is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When *RESET* is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as *RESET* becomes TRUE, independent of *CLK*.

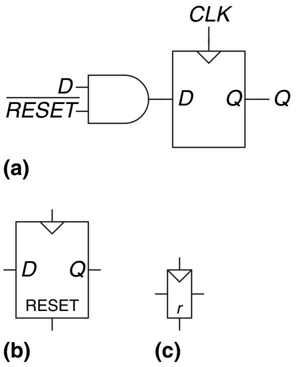


Figure 3.11 **Synchronously resettable flip-flop: (a) schematic, (b, c) symbols**

Figure 3.11(a) shows how to construct a synchronously resettable flip-flop from an ordinary D flip-flop and an AND gate. When $\overline{RESET}$ is FALSE, the AND gate forces a 0 into the input of the flip-flop. When $\overline{RESET}$ is TRUE, the AND gate passes *D* to the flip-flop. In this example, $\overline{RESET}$ is an *active low* signal, meaning that the reset signal performs its function when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high reset signal instead. Figures 3.11(b) and 3.11(c) show symbols for the resettable flip-flop with active high reset.

Asynchronously resettable flip-flops require modifying the internal structure of the flip-flop and are left to you to design in Exercise 3.13; however, they are frequently available to the designer as a standard component.

As you might imagine, settable flip-flops are also occasionally used. They load a 1 into the flip-flop when SET is asserted, and they too come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into *N*-bit registers.

3.2.7 Transistor-Level Latch and Flip-Flop Designs*

Example 3.1 showed that latches and flip-flops require a large number of transistors when built from logic gates. But the fundamental role of a latch is to be transparent or opaque, much like a switch. Recall from

Section 1.7.7 that a transmission gate is an efficient way to build a CMOS switch, so we might expect that we could take advantage of transmission gates to reduce the transistor count.

A compact D latch can be constructed from a single transmission gate, as shown in Figure 3.12(a). When $CLK = 1$ and $\overline{CLK} = 0$, the transmission gate is ON, so D flows to Q and the latch is transparent. When $CLK = 0$ and $\overline{CLK} = 1$, the transmission gate is OFF, so Q is isolated from D and the latch is opaque. This latch suffers from two major limitations:

- *Floating output node:* When the latch is opaque, Q is not held at its value by any gates. Thus Q is called a *floating* or *dynamic* node. After some time, noise and charge leakage may disturb the value of Q.
- *No buffers:* The lack of buffers has caused malfunctions on several commercial chips. A spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when $CLK = 0$. Likewise, a spike on D above V_{DD} can turn on the pMOS transistor even when $CLK = 0$. And the transmission gate is symmetric, so it could be driven backward with noise on Q affecting the input D. The general rule is that neither the input of a transmission gate nor the state node of a sequential circuit should ever be exposed to the outside world, where noise is likely.

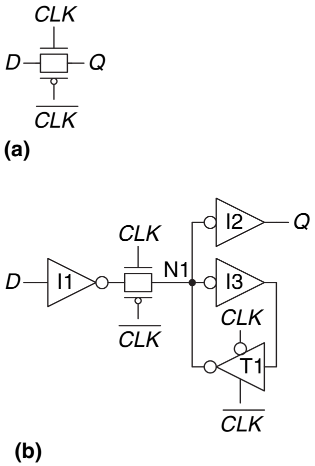


Figure 3.12 **D latch schematic**

Figure 3.12(b) shows a more robust 12-transistor D latch used on modern commercial chips. It is still built around a clocked transmission gate, but it adds inverters I1 and I2 to buffer the input and output. The state of the latch is held on node N1. Inverter I3 and the tristate buffer, T1, provide feedback to turn N1 into a *static node*. If a small amount of noise occurs on N1 while $CLK = 0$, T1 will drive N1 back to a valid logic value.

This circuit assumes CLK and $\overline{CLK}$ are both available. If not, two more transistors are needed for a CLK inverter.

Figure 3.13 shows a D flip-flop constructed from two static latches controlled by $\overline{CLK}$ and CLK. Some redundant internal inverters have been removed, so the flip-flop requires only 20 transistors.

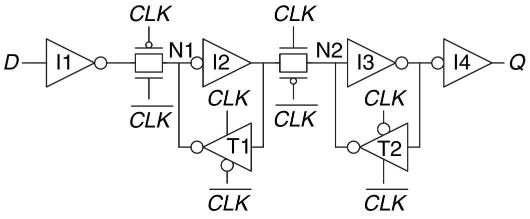


Figure 3.13 **D flip-flop schematic**

3.2.8 Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits. Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when $CLK = 1$, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state. A register is a bank of several D flip-flops that share a common CLK signal.

Example 3.2 FLIP-FLOP AND LATCH COMPARISON

Ben Bitdiddle applies the D and CLK inputs shown in Figure 3.14 to a D latch and a D flip-flop. Help him determine the output, Q, of each device.

Solution: Figure 3.15 shows the output waveforms, assuming a small delay for Q to respond to input changes. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First consider the latch. On the first rising edge of CLK, $D = 0$, so Q definitely becomes 0. Each time D changes while $CLK = 1$, Q also follows. When D changes while $CLK = 0$, it is ignored. Now consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.

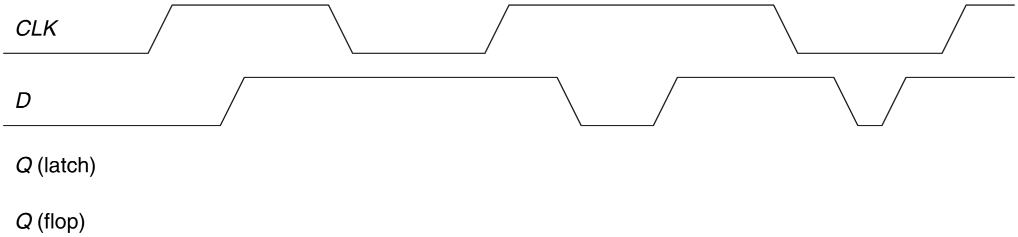


Figure 3.14 Example waveforms

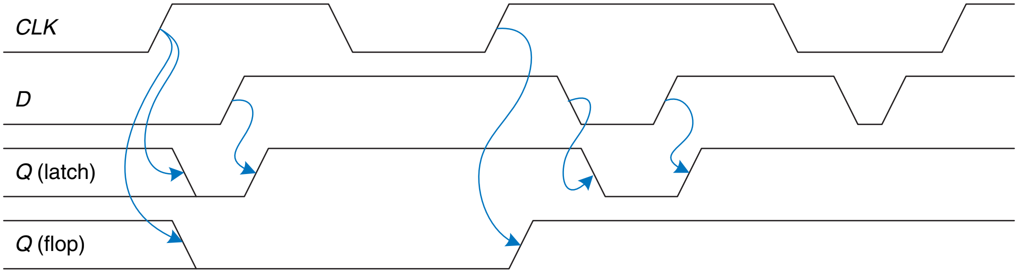


Figure 3.15 Solution waveforms

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational—that is, those whose output cannot be determined simply by looking at the current inputs. Some sequential circuits are just plain kooky. This section begins by examining some of those curious circuits. It then introduces the notion of synchronous sequential circuits and the dynamic discipline. By disciplining ourselves to synchronous sequential circuits, we can develop easy, systematic ways to analyze and design sequential systems.

3.3.1 Some Problematic Circuits

Example 3.3 ASTABLE CIRCUITS

Alyssa P. Hacker encounters three misbegotten inverters who have tied themselves in a loop, as shown in Figure 3.16. The output of the third inverter is *fed back* to the first inverter. Each inverter has a propagation delay of 1 ns. Help Alyssa determine what the circuit does.

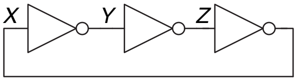


Figure 3.16 **Three-inverter loop**

Solution: Suppose node X is initially 0. Then $Y = 1$, $Z = 0$, and hence $X = 1$, which is inconsistent with our original assumption. The circuit has no stable states and is said to be *unstable* or *astable*. Figure 3.17 shows the behavior of the circuit. If X rises at time 0, Y will fall at 1 ns, Z will rise at 2 ns, and X will fall again at 3 ns. In turn, Y will rise at 4 ns, Z will fall at 5 ns, and X will rise again at 6 ns, and then the pattern will repeat. Each node oscillates between 0 and 1 with a *period* (repetition time) of 6 ns. This circuit is called a *ring oscillator*.

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

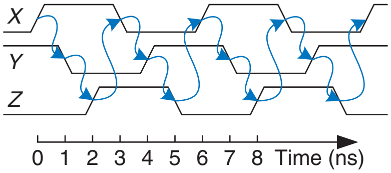


Figure 3.17 **Ring oscillator waveforms**

Example 3.4 RACE CONDITIONS

Ben Bitdiddle designed a new D latch that he claims is better than the one in Figure 3.7 because it uses fewer gates. He has written the truth table to find the

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

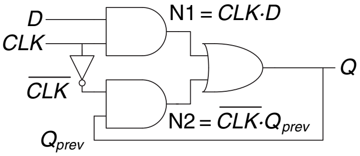


Figure 3.18 **An improved (?) D latch**

output, Q, given the two inputs, D and CLK, and the old state of the latch, Q_{prev}. Based on this truth table, he has derived Boolean equations. He obtains Q_{prev} by feeding back the output, Q. His design is shown in Figure 3.18. Does his latch work correctly, independent of the delays of each gate?

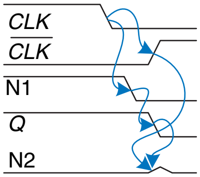


Figure 3.19 **Latch waveforms illustrating race condition**

Solution: Figure 3.19 shows that the circuit has a *race condition* that causes it to fail when certain gates are slower than others. Suppose $CLK = D = 1$. The latch is transparent and passes D through to make $Q = 1$. Now, CLK falls. The latch should remember its old value, keeping $Q = 1$. However, suppose the delay through the inverter from CLK to $\overline{CLK}$ is rather long compared to the delays of the AND and OR gates. Then nodes N1 and Q may both fall before $\overline{CLK}$ rises. In such a case, N2 will never rise, and Q becomes stuck at 0.

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. One circuit may work, while a seemingly identical one built from gates with slightly different delays may not work. Or the circuit may work only at certain temperatures or voltages at which the delays are just right. These malfunctions are extremely difficult to track down.

3.3.2 Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. If inputs are applied to combinational logic, the outputs will always settle to the correct value within a propagation delay. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior. Analyzing such circuits for problems is time-consuming, and many bright people have made mistakes.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

Recall that a circuit is defined by its input and output terminals and its functional and timing specifications. A sequential circuit has a finite set of discrete *states* $\{S_0, S_1, \ldots, S_{k-1}\}$. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The functional specification details the next state and the value of each output for each possible combination of current state and input values. The timing specification consists of an upper bound, t_{pcq}, and a lower bound, t_{ccq}, on the time from the rising edge of the clock until the *output* changes, as well as *setup* and *hold* times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

The rules of *synchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the simplest synchronous sequential circuit. It has one input, *D*, one clock, *CLK*, one output, *Q*, and two states, $\{0, 1\}$. The functional specification for a flip-flop is that the next state is *D* and that the output, *Q*, is the current state, as shown in Figure 3.20.

We often call the current state variable *S* and the next state variable *S*′. In this case, the prime after *S* indicates next state, not inversion. The timing of sequential circuits will be analyzed in Section 3.5.

Two other common types of synchronous sequential circuits are called finite state machines and pipelines. These will be covered later in this chapter.

t_{pcq} stands for the time of propagation from clock to *Q*, where *Q* indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to *Q*. These are analogous to t_{pd} and t_{cd} in combinational logic.

This definition of a synchronous sequential circuit is sufficient, but more restrictive than necessary. For example, in high-performance microprocessors, some registers may receive delayed or gated clocks to squeeze out the last bit of performance or power. Similarly, some microprocessors use latches instead of registers. However, the definition is adequate for all of the synchronous sequential circuits covered in this book and for most commercial digital systems.

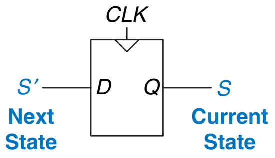


Figure 3.20 **Flip-flop current state and next state**

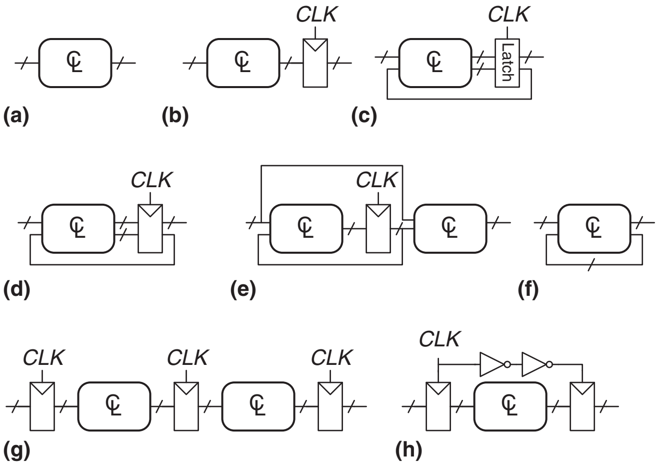


Figure 3.21 **Example circuits**

Example 3.5 SYNCHRONOUS SEQUENTIAL CIRCUITS

Which of the circuits in Figure 3.21 are synchronous sequential circuits?

Solution: Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit, because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines, which are discussed in Section 3.4. (f) is neither combinational nor synchronous sequential, because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline, which we will study in Section 3.6. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3.3.3 Synchronous and Asynchronous Circuits

Asynchronous design in theory is more general than synchronous design, because the timing of the system is not limited by clocked registers. Just as analog circuits are more general than digital circuits because analog circuits can use any voltage, asynchronous circuits are more general than synchronous circuits because they can use any kind of feedback. However, synchronous circuits have proved to be easier to design and use than asynchronous circuits, just as digital are easier than analog circuits. Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous.

Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can improve synchronous circuits too.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, *Gedanken-experiments on Sequential Machines* in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy (1927–2010) published *A Method of Synthesizing Sequential Circuits* in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. They are busy reading about FSMs in their favorite textbook and aren't

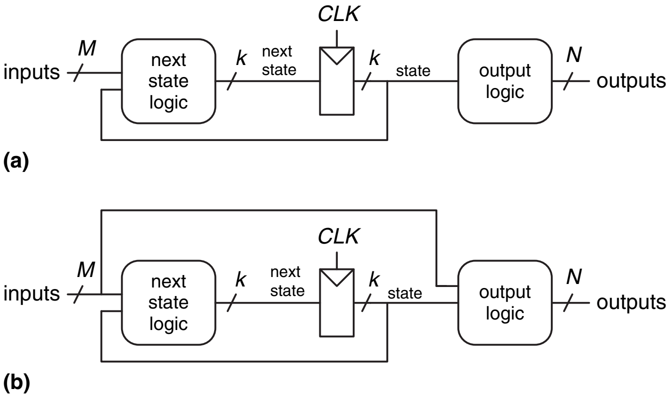


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all the possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long as

Figure 3.23 Campus map

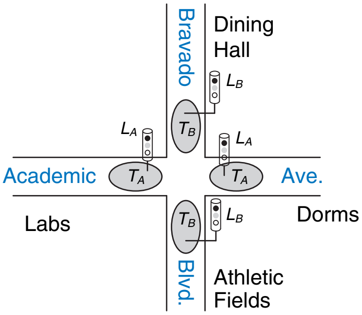
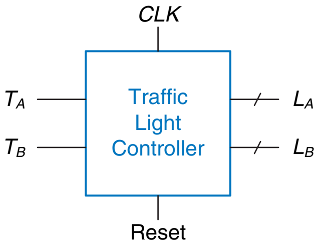


Figure 3.24 Black box view of finite state machine



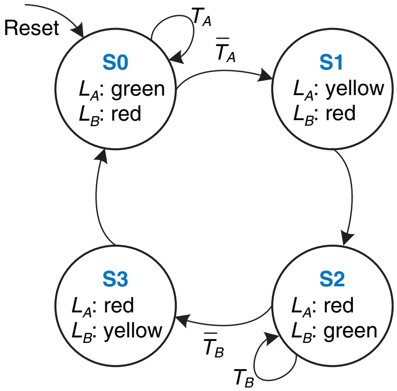


Figure 3.25 State transition diagram

traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates that the system should enter that state upon reset, regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and each output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Notice that states are designated as S0, S1, etc. The subscripted versions, S_0, S_1, etc., refer to the state bits.

Table 3.1 State transition table

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines next state, S', as a function of the current state, S, and the inputs.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$\begin{aligned} S'_1 &= \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B \\ S'_0 &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.1}$$

The equations can be simplified using Karnaugh maps, but often doing it by inspection is easier. For example, the T_B and $\overline{T}_B$ terms in the S'_1 equation are clearly redundant. Thus S'_1 reduces to an XOR operation. Equation 3.2 gives the simplified *next state equations*.

Table 3.4 State transition table with binary encodings

Current State S_1	S_0	Inputs T_A	T_B	Next State S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 **Output table**

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S'_1 &= S_1 \oplus S_0 \\ S'_0 &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.2}$$

Similarly, Ben writes an *output table* (Table 3.5) indicating, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state from the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs from the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows CLK, Reset, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edges of CLK when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram, S

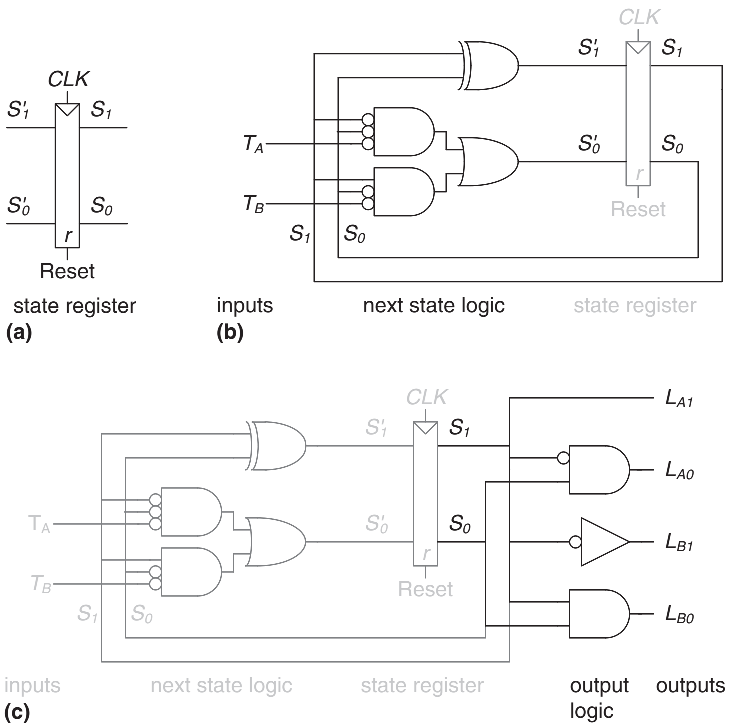

CLK
S'_1
S_1
S'_0
S_0
r
Reset
state register
(a)
T_A
T_B
S_1
S_0
inputs
next state logic
state register
(b)
L_{A1}
L_{A0}
L_{B1}
L_{B0}
inputs
next state logic
state register
output logic
outputs
(c)

Figure 3.26 State machine circuit for traffic light controller

This schematic uses some AND gates with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the non-bubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

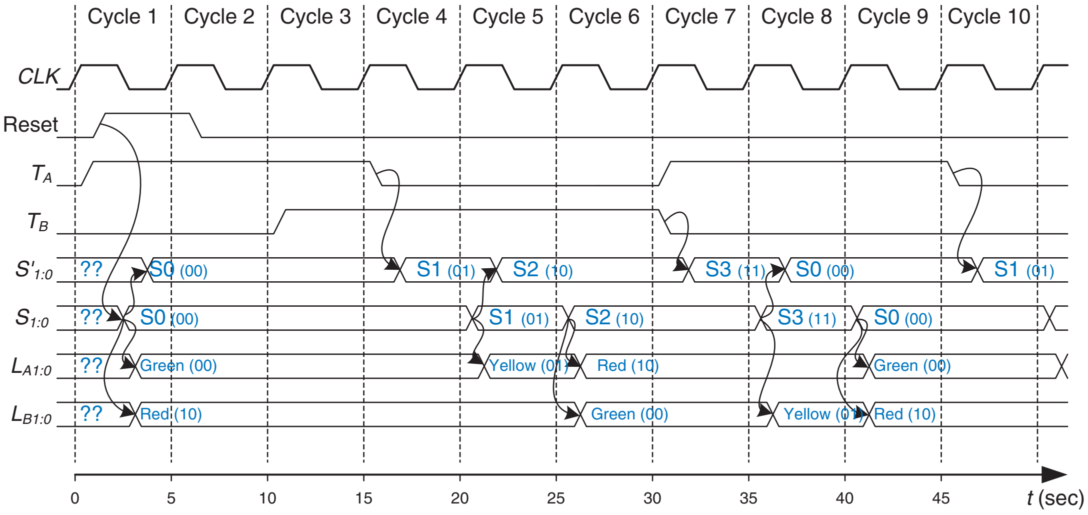
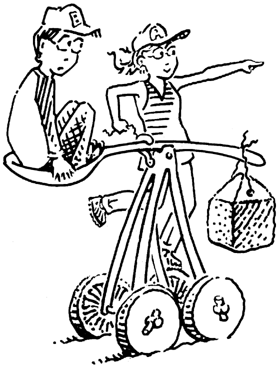


Figure 3.27 Timing diagram for traffic light controller

immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2 in which L_A turns red and L_B turns green. The controller waits in state S2 until all the traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. 5 seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him and Alyssa to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.



3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection, so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding*, as was used in the traffic light controller example, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states only needs $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

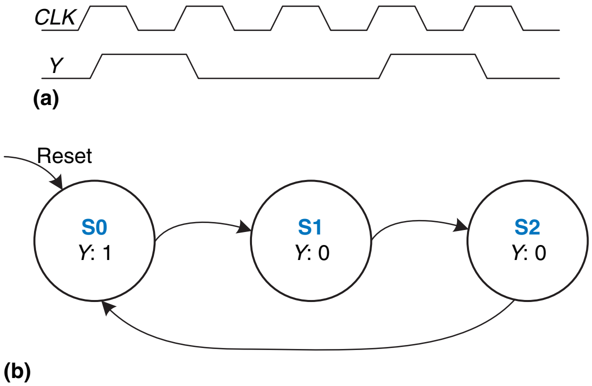


Figure 3.28 Divide-by-3 counter (a) waveform and (b) state transition diagram

Table 3.6 Divide-by-3 counter state transition table

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 Divide-by-3 counter output table

Current State	Output
S0	1
S1	0
S2	0

Solution: Tables 3.6 and 3.7 show the abstract state transition and output tables before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S'_1 &= \overline{S}_1 S_0 \\ S'_0 &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S'_2 &= S_1 \\ S'_1 &= S_0 \\ S'_0 &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S'_0. Also observe that the one-hot encoding requires both settable (*s*) and resettable (*r*) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which *K* states are represented with *K* bits, exactly one of which is FALSE.

Table 3.8 One-hot and binary encodings for divide-by-3 counter

State	One-Hot Encoding S_2	S_1	S_0	Binary Encoding S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Table 3.9 State transition table with binary encoding

Current State S_1	S_0	Next State S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State S_2	S_1	S_0	Next State S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

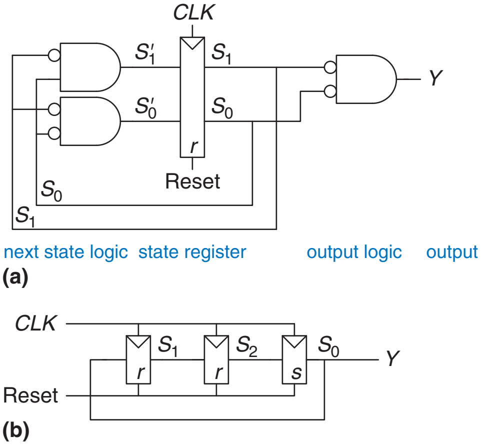

CLK
S'_1
S_1
S'_0
S_0
r
Reset
S_0
S_1
Y
next state logic
state register
output logic
output
(a)
CLK
S_1
S_2
S_0
r
r
s
Y
Reset
(b)

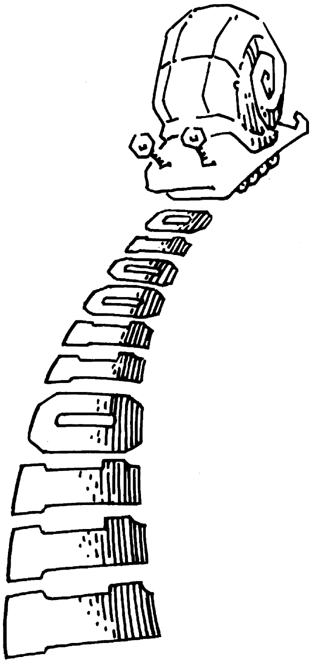
Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).

Example 3.7 MOORE VERSUS MEALY MACHINES



Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are 01. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as Alyssa's snail crawls along the sequence 0100110111.

Solution: The Moore machine requires three states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S2 to S1 when the input is 0?

In comparison, the Mealy machine requires only two states, as shown in Figure 3.30(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables for the Moore machine. The Moore machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, and S2 = 10. Tables 3.13 and 3.14 rewrite the state transition and output tables with these encodings.

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that state 11 does not exist. Thus, the corresponding next state and output for the non-existent state are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$\begin{aligned} S'_1 &= S_0 A \\ S'_0 &= \overline{A} \end{aligned} \tag{3.8}$$

$$Y = S_1 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires only one bit of state. Consider using a binary state encoding: S0 = 0 and S1 = 1. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$S'_0 = \overline{A} \tag{3.10}$$

$$Y = S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31. The timing diagrams for each machine are shown in Figure 3.32 (see page 135). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

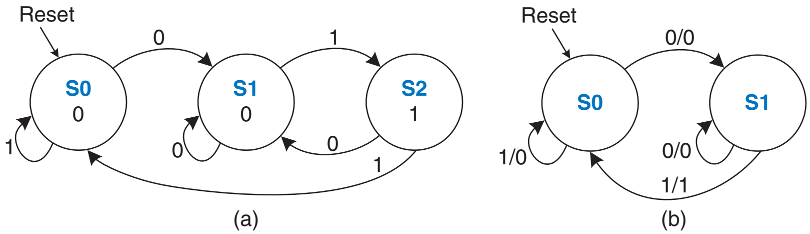


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Table 3.11 Moore state transition table

Current State *S*	Input *A*	Next State *S′*
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Table 3.12 Moore output table

Current State *S*	Output *Y*
S0	0
S1	0
S2	1

Table 3.13 Moore state transition table with state encodings

Current State S_1	Current State S_0	Input A	Next State S'_1	Next State S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Table 3.14 Moore output table with state encodings

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Table 3.15 Mealy state transition and output table

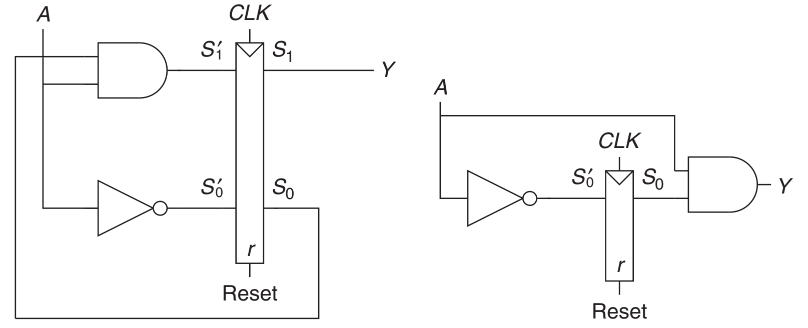
Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Table 3.16 Mealy state transition and output table with state encodings

Current State S_0	Input A	Next State S'_0	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

3.4.4 Factoring State Machines

Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.



(a) (b)

Figure 3.31 **FSM schematics for (a) Moore and (b) Mealy machines**

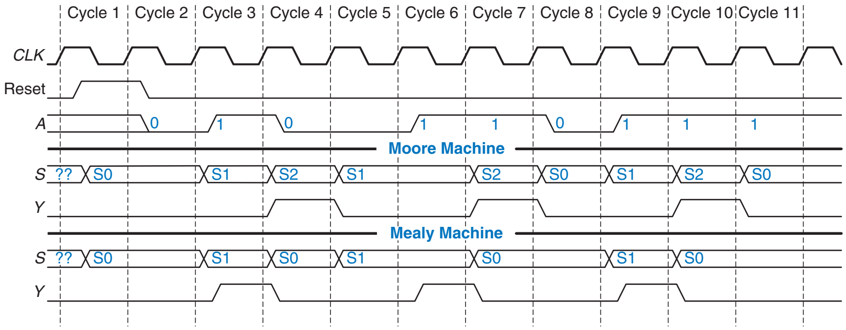


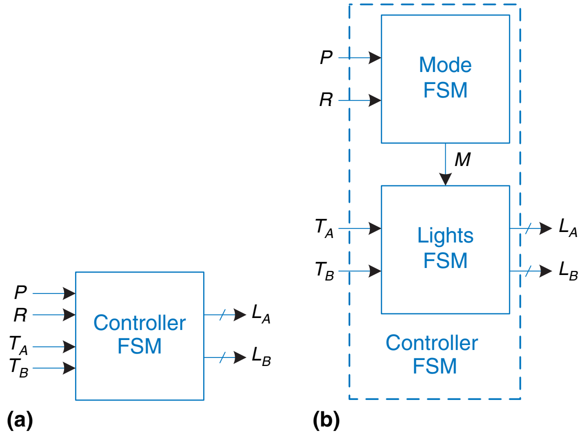
Figure 3.32 **Timing diagrams for Moore and Mealy machines**

Example 3.8 UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

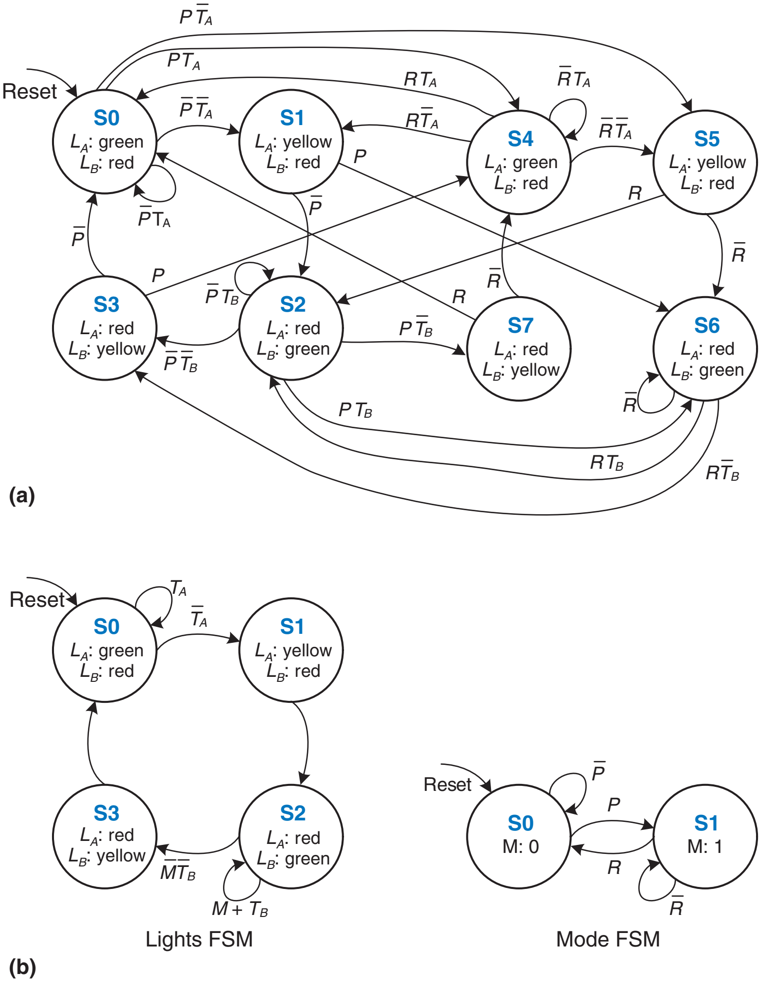
First, sketch a state transition diagram for a single FSM, as shown in Figure 3.33(a). Then, sketch the state transition diagrams for two interacting FSMs, as shown in Figure 3.33(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors, T_A and T_B.

Solution: Figure 3.34(a) shows the single FSM design. States S0 to S3 handle normal mode. States S4 to S7 handle parade mode. The two halves of the diagram are almost identical, but in parade mode, the FSM remains in S6 with a green light on Bravado Blvd. The P and R inputs control movement between these two halves. The FSM is messy and tedious to design. Figure 3.34(b) shows the factored FSM design. The mode FSM has two states to track whether the lights are in normal or parade mode. The Lights FSM is modified to remain in S2 while M is TRUE.



P
R
T_A
T_B
Controller FSM
L_A
L_B
(a)
P
R
Mode FSM
M
T_A
T_B
Lights FSM
L_A
L_B
Controller FSM
(b)

Figure 3.33 (a) single and (b) factored designs for modified traffic light controller FSM



Reset
S0 L_A: green L_B: red
S1 L_A: yellow L_B: red
S2 L_A: red L_B: green
S3 L_A: red L_B: yellow
S4 L_A: green L_B: red
S5 L_A: yellow L_B: red
S6 L_A: red L_B: green
S7 L_A: red L_B: yellow
$P\overline{T_A}$
PT_A
RT_A
$\overline{R}T_A$
$\overline{P}\,\overline{T_A}$
$R\overline{T_A}$
$\overline{R}\,\overline{T_A}$
$\overline{P}T_A$
P
$\overline{P}$
R
$\overline{R}$
$\overline{P}T_B$
$P\overline{T_B}$
$\overline{P}\,\overline{T_B}$
PT_B
RT_B
$R\overline{T_B}$
(a)
T_A
$\overline{T_A}$
$\overline{M}\,\overline{T_B}$
$M + T_B$
Lights FSM
S0 M: 0
S1 M: 1
$\overline{P}$
P
R
$\overline{R}$
Mode FSM
(b)

Figure 3.34 State transition diagrams: (a) unfactored, (b) factored

3.4.5 Deriving an FSM from a Schematic

Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design. This process can be necessary, for example, when taking on an incompletely documented project or reverse engineering somebody else's system.

- Examine circuit, stating inputs, outputs, and state bits.
- Write next state and output equations.
- Create next state and output tables.
- Reduce the next state table to eliminate unreachable states.
- Assign each valid state bit combination a name.
- Rewrite next state and output tables with state names.
- Draw state transition diagram.
- State in words what the FSM does.

In the final step, be careful to succinctly describe the overall purpose and function of the FSM—do not simply restate each transition of the state transition diagram.

Example 3.9 DERIVING AN FSM FROM ITS CIRCUIT

Alyssa P. Hacker arrives home, but her keypad lock has been rewired and her old code no longer works. A piece of paper is taped to it showing the circuit diagram in Figure 3.35. Alyssa thinks the circuit could be a finite state machine and decides to derive the state transition diagram to see if it helps her get in the door.

Solution: Alyssa begins by examining the circuit. The input is $A_{1:0}$ and the output is *Unlock*. The state bits are already labeled in Figure 3.35. This is a Moore

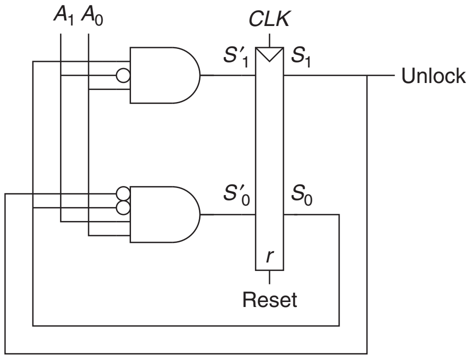


Figure 3.35 Circuit of found FSM for Example 3.9

machine because the output depends only on the state bits. From the circuit, she writes down the next state and output equations directly:

$$\begin{aligned} S'_1 &= S_0\overline{A_1}A_0 \\ S'_0 &= \overline{S_1}\,\overline{S_0}A_1A_0 \\ Unlock &= S_1 \end{aligned} \tag{3.12}$$

Next, she writes down the next state and output tables from the equations, as shown in Tables 3.17 and 3.18, first placing 1's in the tables as indicated by Equation 3.12. She places 0's everywhere else.

Alyssa reduces the table by removing unused states and combining rows using don't cares. The $S_{1:0} = 11$ state is never listed as a possible next state in Table 3.17, so rows with this current state are removed. For current state $S_{1:0} = 10$, the next

Table 3.17 **Next state table derived from circuit in Figure 3.35**

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	**1**
0	1	0	0	0	0
0	1	0	1	**1**	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	**1**	0
1	1	1	0	0	0
1	1	1	1	0	0

Table 3.18 **Output table derived from circuit in Figure 3.35**

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	**1**
1	1	**1**

Table 3.19 **Reduced next state table**

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S'_1	Next State S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Table 3.20 **Reduced output table**

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1

Table 3.21 **Symbolic next state table**

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Table 3.22 **Symbolic output table**

Current State S	Output *Unlock*
S0	0
S1	0
S2	1

state is always $S_{1:0} = 00$, independent of the inputs, so don't cares are inserted for the inputs. The reduced tables are shown in Tables 3.19 and 3.20.

She assigns names to each state bit combination: S0 is $S_{1:0} = 00$, S1 is $S_{1:0} = 01$, and S2 is $S_{1:0} = 10$. Tables 3.21 and 3.22 show the next state and output tables with state names.

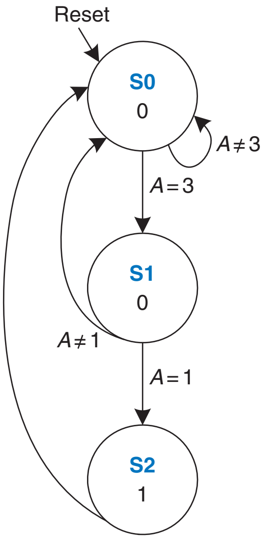


Figure 3.36 **State transition diagram of found FSM from Example 3.9**

Alyssa writes down the state transition diagram shown in Figure 3.36 using Tables 3.21 and 3.22. By inspection, she can see that the finite state machine unlocks the door only after detecting an input value, $A_{1:0}$, of three followed by an input value of one. The door is then locked again. Alyssa tries this code on the door key pad and the door opens!

3.4.6 FSM Review

Finite state machines are a powerful way to systematically design sequential circuits from a written specification. Use the following procedure to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.
- For a Moore machine:
  - Write a state transition table.
  - Write an output table.
- For a Mealy machine:
  - Write a combined state transition and output table.
- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

We will repeatedly use FSMs to design complex digital systems throughout this book.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?

This problem is similar to that faced by a camera when snapping a picture. Imagine photographing a frog jumping from a lily pad into the lake. If you take the picture before the jump, you will see a frog on a lily pad. If you take the picture after the jump, you will see ripples in the water. But if you take it just as the frog jumps, you may see a blurred image of the frog stretching from the lily pad into the water. A camera is characterized by its *aperture time*, during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a *setup* time and a *hold* time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time. By taking advantage of the dynamic discipline, we can think of time in discrete units called clock cycles, just as we think of signal levels as discrete 1's and 0's. A signal may glitch and oscillate wildly for some bounded amount of time. Under the dynamic discipline, we are concerned only about its final value at the end of the clock cycle, after it has settled to a stable value. Hence, we can simply write $A[n]$, the value of signal A at the end of the *nth* clock cycle, where n is an integer, rather than $A(t)$, the value of A at some instant t, where t is any real number.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not reach all flip-flops at precisely the same time. This variation in time, called clock skew, further increases the necessary clock period.

Sometimes it is impossible to satisfy the dynamic discipline, especially when interfacing with the real world. For example, consider a circuit with an input coming from a button. A monkey might press the button just as the clock rises. This can result in a phenomenon called metastability, where the flip-flop captures a value partway between 0 and 1 that can take an unlimited amount of time to resolve into a good logic value. The solution to such asynchronous inputs is to use a synchronizer, which has a very small (but nonzero) probability of producing an illegal logic value.

We expand on all of these ideas in the rest of this section.

3.5.1 The Dynamic Discipline

So far, we have focused on the functional specification of sequential circuits. Recall that a synchronous sequential circuit, such as a flip-flop or FSM, also has a timing specification, as illustrated in Figure 3.37. When the clock rises, the output (or outputs) may start to change after the clock-to-Q *contamination delay*, t_{ccq}, and must definitely settle to the final value within the clock-to-Q *propagation delay*, t_{pcq}. These represent the fastest and slowest delays through the circuit, respectively. For the circuit to sample its input correctly, the input (or inputs) must have stabilized at least some *setup time*, t_{setup}, before the rising edge of the clock and must remain stable for at least some *hold time*, t_{hold}, after the rising edge of the clock. The sum of the setup and hold times is called the *aperture time* of the circuit, because it is the total time for which the input must remain stable.

The *dynamic discipline* states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge. By imposing this requirement, we guarantee that the flip-flops sample signals while they are not changing. Because we are concerned only about the final values of the inputs at the time they are sampled, we can treat signals as discrete in time as well as in logic levels.

3.5.2 System Timing

In the three decades from when one of the authors' families bought an Apple II+ computer to the present time of writing, microprocessor clock frequencies have increased from 1 MHz to several GHz, a factor of more than 1000. This speedup partially explains the revolutionary changes computers have made in society.

The *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*. All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time. Frequency is measured in units of Hertz (Hz), or cycles per second: 1 megahertz (MHz) = 10^6 Hz, and 1 gigahertz (GHz) = 10^9 Hz.

Figure 3.38(a) illustrates a generic path in a synchronous sequential circuit whose clock period we wish to calculate. On the rising edge of the clock, register R1 produces output (or outputs) $Q1$. These signals enter a block of combinational logic, producing $D2$, the input (or inputs) to register R2. The timing diagram in Figure 3.38(b) shows that each output signal may start to change a contamination delay after its input

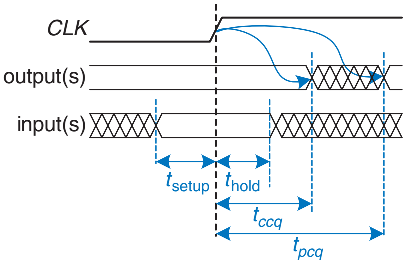


Figure 3.37 **Timing specification for synchronous sequential circuit**

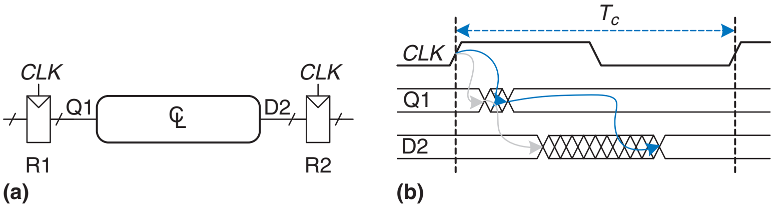


Figure 3.38 **Path between registers and timing diagram**

changes and settles to the final value within a propagation delay after its input settles. The gray arrows represent the contamination delay through R1 and the combinational logic, and the blue arrows represent the propagation delay through R1 and the combinational logic. We analyze the timing constraints with respect to the setup and hold time of the second register, R2.

Setup Time Constraint

Figure 3.39 is the timing diagram showing only the maximum delay through the path, indicated by the blue arrows. To satisfy the setup time of R2, $D2$ must settle no later than the setup time before the next clock edge. Hence, we find an equation for the minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \tag{3.13}$$

In commercial designs, the clock period is often dictated by the Director of Engineering or by the marketing department (to ensure a competitive product). Moreover, the flip-flop clock-to-Q propagation delay and setup time, t_{pcq} and t_{setup}, are specified by the manufacturer. Hence, we rearrange Equation 3.13 to solve for the maximum propagation delay through the combinational logic, which is usually the only variable under the control of the individual designer.

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}) \tag{3.14}$$

The term in parentheses, $t_{pcq} + t_{\text{setup}}$, is called the *sequencing overhead*. Ideally, the entire cycle time T_c would be available for useful

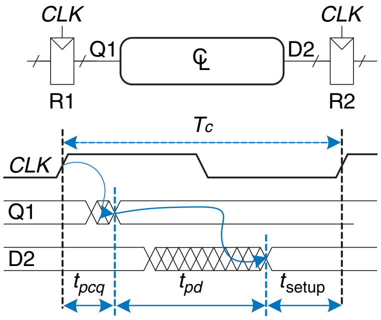


Figure 3.39 **Maximum delay for setup time constraint**

computation in the combinational logic, t_{pd}. However, the sequencing overhead of the flip-flop cuts into this time. Equation 3.14 is called the *setup time constraint* or *max-delay constraint*, because it depends on the setup time and limits the maximum delay through combinational logic.

If the propagation delay through the combinational logic is too great, $D2$ may not have settled to its final value by the time R2 needs it to be stable and samples it. Hence, R2 may sample an incorrect result or even an illegal logic level, a level in the forbidden region. In such a case, the circuit will malfunction. The problem can be solved by increasing the clock period or by redesigning the combinational logic to have a shorter propagation delay.

Hold Time Constraint

The register R2 in Figure 3.38(a) also has a *hold time constraint*. Its input, $D2$, must not change until some time, t_{hold}, after the rising edge of the clock. According to Figure 3.40, $D2$ might change as soon as $t_{ccq} + t_{cd}$ after the rising edge of the clock. Hence, we find

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} \tag{3.15}$$

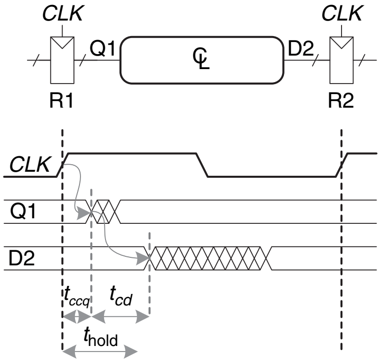
Again, t_{ccq} and t_{hold} are characteristics of the flip-flop that are usually outside the designer's control. Rearranging, we can solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq} \tag{3.16}$$

Equation 3.16 is also called the *hold time constraint* or *min-delay constraint* because it limits the minimum delay through combinational logic.

We have assumed that any logic elements can be connected to each other without introducing timing problems. In particular, we would expect that two flip-flops may be directly cascaded as in Figure 3.41 without causing hold time problems.

Figure 3.40 Minimum delay for hold time constraint



In such a case, $t_{cd} = 0$ because there is no combinational logic between flip-flops. Substituting into Equation 3.16 yields the requirement that

$$t_{\text{hold}} \leq t_{ccq} \tag{3.17}$$

In other words, a reliable flip-flop must have a hold time shorter than its contamination delay. Often, flip-flops are designed with $t_{\text{hold}} = 0$, so that Equation 3.17 is always satisfied. Unless noted otherwise, we will usually make that assumption and ignore the hold time constraint in this book.

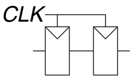


Figure 3.41 **Back-to-back flip-flops**

Nevertheless, hold time constraints are critically important. If they are violated, the only solution is to increase the contamination delay through the logic, which requires redesigning the circuit. Unlike setup time constraints, they cannot be fixed by adjusting the clock period. Redesigning an integrated circuit and manufacturing the corrected design takes months and millions of dollars in today's advanced technologies, so *hold time violations* must be taken extremely seriously.

Putting It All Together

Sequential circuits have setup and hold time constraints that dictate the maximum and minimum delays of the combinational logic between flip-flops. Modern flip-flops are usually designed so that the minimum delay through the combinational logic is 0—that is, flip-flops can be placed back-to-back. The maximum delay constraint limits the number of consecutive gates on the critical path of a high-speed circuit, because a high clock frequency means a short clock period.

Example 3.10 TIMING ANALYSIS

Ben Bitdiddle designed the circuit in Figure 3.42. According to the data sheets for the components he is using, flip-flops have a clock-to-Q contamination delay of 30 ps and a propagation delay of 80 ps. They have a setup time of 50 ps and a hold time of 60 ps. Each logic gate has a propagation delay of 40 ps and a

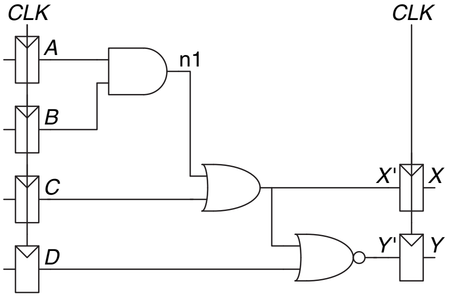


Figure 3.42 **Sample circuit for timing analysis**

contamination delay of 25 ps. Help Ben determine the maximum clock frequency and whether any hold time violations could occur. This process is called *timing analysis*.

Solution: Figure 3.43(a) shows waveforms illustrating when the signals might change. The inputs, A to D, are registered, so they only change shortly after CLK rises.

The critical path occurs when $B = 1$, $C = 0$, $D = 0$, and A rises from 0 to 1, triggering n1 to rise, X' to rise, and Y' to fall, as shown in Figure 3.43(b). This path involves three gate delays. For the critical path, we assume that each gate requires its full propagation delay. Y' must setup before the next rising edge of the CLK. Hence, the minimum cycle time is

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250\text{ps} \tag{3.18}$$

The maximum clock frequency is $f_c = 1/T_c = 4$ GHz.

A short path occurs when $A = 0$ and C rises, causing X' to rise, as shown in Figure 3.43(c). For the short path, we assume that each gate switches after only a contamination delay. This path involves only one gate delay, so it may occur after $t_{ccq} + t_{cd} = 30 + 25 = 55$ ps. But recall that the flip-flop has a hold time of

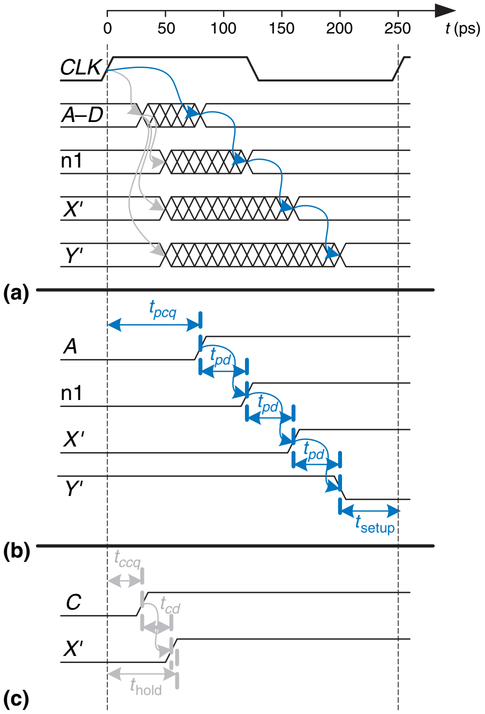


Figure 3.43 **Timing diagram: (a) general case, (b) critical path, (c) short path**

60 ps, meaning that X' must remain stable for 60 ps after the rising edge of *CLK* for the flip-flop to reliably sample its value. In this case, $X' = 0$ at the first rising edge of *CLK*, so we want the flip-flop to capture $X = 0$. Because X' did not hold stable long enough, the actual value of X is unpredictable. The circuit has a hold time violation and may behave erratically at any clock frequency.

Example 3.11 FIXING HOLD TIME VIOLATIONS

Alyssa P. Hacker proposes to fix Ben's circuit by adding buffers to slow down the short paths, as shown in Figure 3.44. The buffers have the same delays as other gates. Help her determine the maximum clock frequency and whether any hold time problems could occur.

Solution: Figure 3.45 shows waveforms illustrating when the signals might change. The critical path from A to Y is unaffected, because it does not pass through any buffers. Therefore, the maximum clock frequency is still 4 GHz. However, the short paths are slowed by the contamination delay of the buffer. Now X' will not change until $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ ps. This is after the 60 ps hold time has elapsed, so the circuit now operates correctly.

This example had an unusually long hold time to illustrate the point of hold time problems. Most flip-flops are designed with $t_{\text{hold}} < t_{ccq}$ to avoid such problems.

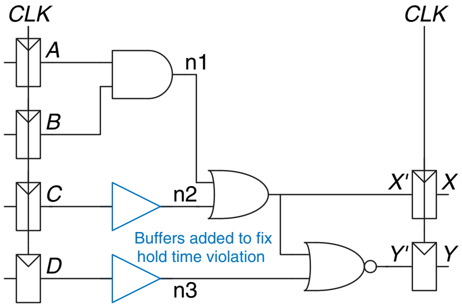


Figure 3.44 Corrected circuit to fix hold time problem

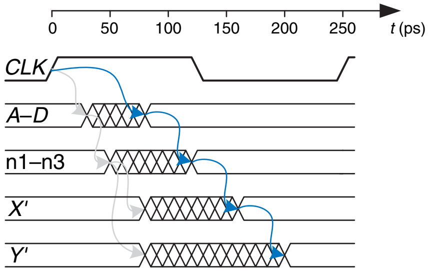


Figure 3.45 Timing diagram with buffers to fix hold time problem

However, some high-performance microprocessors, including the Pentium 4, use an element called a *pulsed latch* in place of a flip-flop. The pulsed latch behaves like a flip-flop but has a short clock-to-Q delay and a long hold time. In general, adding buffers can usually, but not always, solve hold time problems without slowing the critical path.

3.5.3 Clock Skew*

In the previous analysis, we assumed that the clock reaches all registers at exactly the same time. In reality, there is some variation in this time. This variation in clock edges is called *clock skew*. For example, the wires from the clock source to different registers may be of different lengths, resulting in slightly different delays, as shown in Figure 3.46. Noise also results in different delays. Clock gating, described in Section 3.2.5, further delays the clock. If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks. In Figure 3.46, *CLK*2 is *early* with respect to *CLK*1, because the clock wire between the two registers follows a scenic route. If the clock had been routed differently, *CLK*1 might have been early instead. When doing timing analysis, we consider the worst-case scenario, so that we can guarantee that the circuit will work under all circumstances.

Figure 3.47 adds skew to the timing diagram from Figure 3.38. The heavy clock line indicates the latest time at which the clock signal might reach any register; the hashed lines show that the clock might arrive up to t_{skew} earlier.

First, consider the setup time constraint shown in Figure 3.48. In the worst case, R1 receives the latest skewed clock and R2 receives the earliest skewed clock, leaving as little time as possible for data to propagate between the registers.

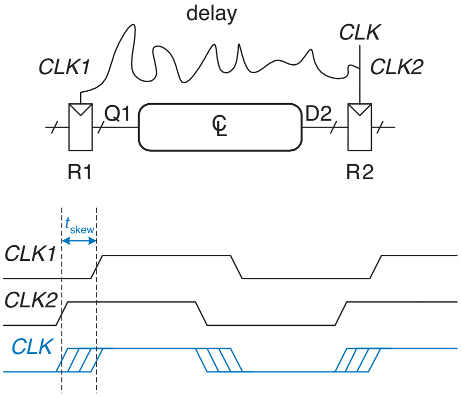


Figure 3.46 Clock skew caused by wire delay

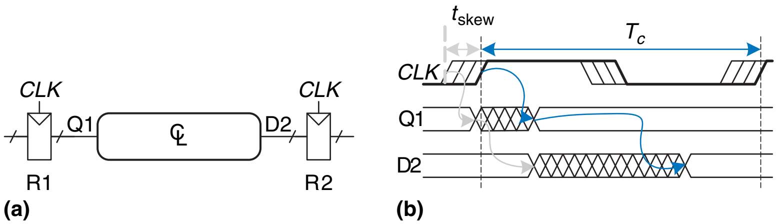


Figure 3.47 Timing diagram with clock skew

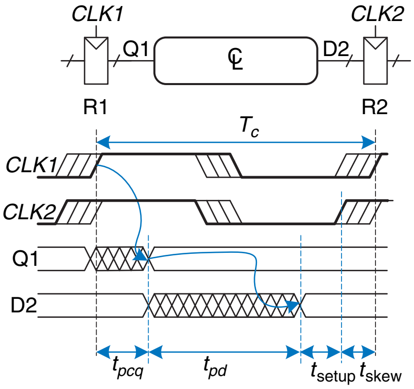


Figure 3.48 Setup time constraint with clock skew

The data propagates through the register and combinational logic and must setup before R2 samples it. Hence, we conclude that

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \tag{3.19}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}} + t_{\text{skew}}) \tag{3.20}$$

Next, consider the hold time constraint shown in Figure 3.49. In the worst case, R1 receives an early skewed clock, *CLK*1, and R2 receives a late skewed clock, *CLK*2. The data zips through the register and combinational logic but must not arrive until a hold time after the late clock. Thus, we find that

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} \tag{3.21}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \tag{3.22}$$

In summary, clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic. Even if $t_{\text{hold}} = 0$, a pair of back-to-back flip-flops will violate Equation 3.22 if $t_{\text{skew}} > t_{ccq}$. To prevent

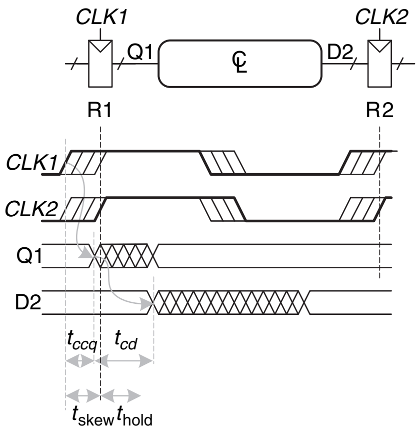


Figure 3.49 **Hold time constraint with clock skew**

serious hold time failures, designers must not permit too much clock skew. Sometimes flip-flops are intentionally designed to be particularly slow (i.e., large t_{ccq}), to prevent hold time problems even when the clock skew is substantial.

Example 3.12 TIMING ANALYSIS WITH CLOCK SKEW

Revisit Example 3.10 and assume that the system has 50 ps of clock skew.

Solution: The critical path remains the same, but the setup time is effectively increased by the skew. Hence, the minimum cycle time is

$$\begin{aligned} T_c &\geq t_{pcq} + 3t_{pd} + t_{\text{setup}} + t_{\text{skew}} \\ &= 80 + 3\times 40 + 50 + 50 = 300\text{ps} \end{aligned} \tag{3.23}$$

The maximum clock frequency is $f_c = 1/T_c = 3.33$ GHz.

The short path also remains the same at 55 ps. The hold time is effectively increased by the skew to $60 + 50 = 110$ ps, which is much greater than 55 ps. Hence, the circuit will violate the hold time and malfunction at any frequency. The circuit violated the hold time constraint even without skew. Skew in the system just makes the violation worse.

Example 3.13 FIXING HOLD TIME VIOLATIONS

Revisit Example 3.11 and assume that the system has 50 ps of clock skew.

Solution: The critical path is unaffected, so the maximum clock frequency remains 3.33 GHz.

The short path increases to 80 ps. This is still less than $t_{\text{hold}} + t_{\text{skew}} = 110$ ps, so the circuit still violates its hold time constraint.

To fix the problem, even more buffers could be inserted. Buffers would need to be added on the critical path as well, reducing the clock frequency. Alternatively, a better flip-flop with a shorter hold time might be used.

3.5.4 Metastability

As noted earlier, it is not always possible to guarantee that the input to a sequential circuit is stable during the aperture time, especially when the input arrives from the external world. Consider a button connected to the input of a flip-flop, as shown in Figure 3.50. When the button is not pressed, $D = 0$. When the button is pressed, $D = 1$. A monkey presses the button at some random time relative to the rising edge of CLK. We want to know the output Q after the rising edge of CLK. In Case I, when the button is pressed much before CLK, $Q = 1$. In Case II, when the button is not pressed until long after CLK, $Q = 0$. But in Case III, when the button is pressed sometime between t_{setup} before CLK and t_{hold} after CLK, the input violates the dynamic discipline and the output is undefined.

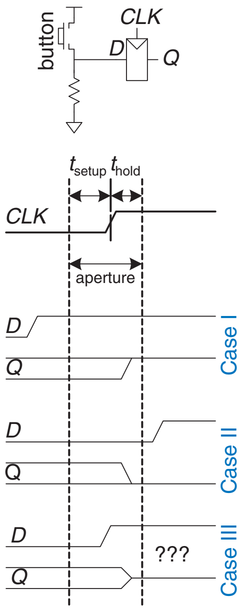


Figure 3.50 Input changing before, after, or during aperture

Metastable State

When a flip-flop samples an input that is changing during its aperture, the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. This is called a *metastable* state. Eventually, the flip-flop will resolve the output to a *stable state* of either 0 or 1. However, the *resolution time* required to reach the stable state is unbounded.

The metastable state of a flip-flop is analogous to a ball on the summit of a hill between two valleys, as shown in Figure 3.51. The two valleys are stable states, because a ball in the valley will remain there as long as it is not disturbed. The top of the hill is called metastable because the ball would remain there if it were perfectly balanced. But because nothing is perfect, the ball will eventually roll to one side or the other. The time required for this change to occur depends on how nearly well balanced the ball originally was. Every bistable device has a metastable state between the two stable states.

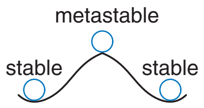


Figure 3.51 Stable and metastable states

Resolution Time

If a flip-flop input changes at a random time during the clock cycle, the resolution time, t_{res}, required to resolve to a stable state is also a random variable. If the input changes outside the aperture, then $t_{res} = t_{pcq}$. But if the input happens to change within the aperture, t_{res} can be substantially longer. Theoretical and experimental analyses (see Section 3.5.6) have shown that the probability that the resolution time, t_{res}, exceeds some arbitrary time, t, decreases exponentially with t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3.24}$$

where T_c is the clock period, and T_0 and τ are characteristic of the flip-flop. The equation is valid only for t substantially longer than t_{pcq}.

Intuitively, T_0/T_c describes the probability that the input changes at a bad time (i.e., during the aperture time); this probability decreases with the cycle time, T_c. τ is a time constant indicating how fast the flip-flop moves away from the metastable state; it is related to the delay through the cross-coupled gates in the flip-flop.

In summary, if the input to a bistable device such as a flip-flop changes during the aperture time, the output may take on a metastable value for some time before resolving to a stable 0 or 1. The amount of time required to resolve is unbounded, because for any finite time, t, the probability that the flip-flop is still metastable is nonzero. However, this probability drops off exponentially as t increases. Therefore, if we wait long enough, much longer than t_{pcq}, we can expect with exceedingly high probability that the flip-flop will reach a valid logic level.

3.5.5 Synchronizers

Asynchronous inputs to digital systems from the real world are inevitable. Human input is asynchronous, for example. If handled carelessly, these asynchronous inputs can lead to metastable voltages within the system, causing erratic system failures that are extremely difficult to track down and correct. The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small. "Sufficiently" depends on the context. For a cell phone, perhaps one failure in 10 years is acceptable, because the user can always turn the phone off and back on if it locks up. For a medical device, one failure in the expected life of the universe (10^{10} years) is a better target. To guarantee good logic levels, all asynchronous inputs should be passed through *synchronizers*.

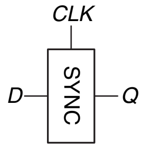


Figure 3.52 Synchronizer symbol

A synchronizer, shown in Figure 3.52, is a device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time; the output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.

Figure 3.53 shows a simple way to build a synchronizer out of two flip-flops. F1 samples D on the rising edge of CLK. If D is changing at that time, the output D2 may be momentarily metastable. If the clock

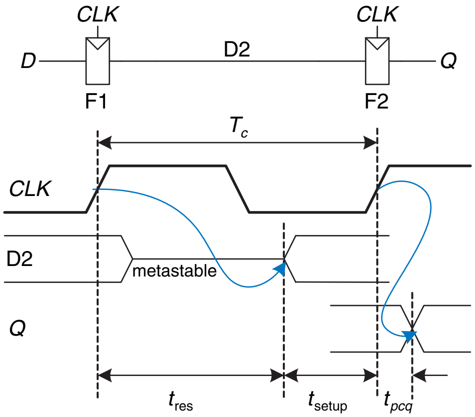


Figure 3.53 Simple synchronizer

period is long enough, D2 will, with high probability, resolve to a valid logic level before the end of the period. F2 then samples D2, which is now stable, producing a good output Q.

We say that a synchronizer *fails* if Q, the output of the synchronizer, becomes metastable. This may happen if $D2$ has not resolved to a valid level by the time it must setup at F2—that is, if $t_{res} > T_c - t_{\text{setup}}$. According to Equation 3.24, the probability of failure for a single input change at a random time is

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.25}$$

The probability of failure, $P(\text{failure})$, is the probability that the output Q will be metastable upon a single change in D. If D changes once per second, the probability of failure per second is just $P(\text{failure})$. However, if D changes N times per second, the probability of failure per second is N times as great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.26}$$

System reliability is usually measured in *mean time between failures* (*MTBF*). As the name suggests, MTBF is the average amount of time between failures of the system. It is the reciprocal of the probability that the system will fail in any given second

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{\frac{T_c - t_{\text{setup}}}{\tau}}}{NT_0} \tag{3.27}$$

Equation 3.27 shows that the MTBF improves exponentially as the synchronizer waits for a longer time, T_c. For most systems, a synchronizer that waits for one clock cycle provides a safe MTBF. In exceptionally high-speed systems, waiting for more cycles may be necessary.

Example 3.14 SYNCHRONIZER FOR FSM INPUT

The traffic light controller FSM from Section 3.4.1 receives asynchronous inputs from the traffic sensors. Suppose that a synchronizer is used to guarantee stable inputs to the controller. Traffic arrives on average 0.2 times per second. The flip-flops in the synchronizer have the following characteristics: $\tau = 200$ ps, $T_0 = 150$ ps, and $t_{\text{setup}} = 500$ ps. How long must the synchronizer clock period be for the MTBF to exceed 1 year?

Solution: 1 year $\approx \pi \times 10^7$ seconds. Solve Equation 3.27.

$$\pi \times 10^7 = \frac{T_c e^{\frac{T_c - 500\times10^{-12}}{200\times10^{-12}}}}{(0.2)(150 \times 10^{-12})} \tag{3.28}$$

This equation has no closed form solution. However, it is easy enough to solve by guess and check. In a spreadsheet, try a few values of T_c and calculate the MTBF until discovering the value of T_c that gives an MTBF of 1 year: $T_c = 3.036$ ns.

3.5.6 Derivation of Resolution Time*

Equation 3.24 can be derived using a basic knowledge of circuit theory, differential equations, and probability. This section can be skipped if you are not interested in the derivation or if you are unfamiliar with the mathematics.

A flip-flop output will be metastable after some time, t, if the flip-flop samples a changing input (causing a metastable condition) and the output does not resolve to a valid level within that time after the clock edge. Symbolically, this can be expressed as

$$P(t_{res} > t) = P(\text{samples changing input}) \times P(\text{unresolved}) \tag{3.29}$$

We consider each probability term individually. The asynchronous input signal switches between 0 and 1 in some time, t_{switch}, as shown in Figure 3.54. The probability that the input changes during the aperture around the clock edge is

$$P(\text{samples changing input}) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{T_c} \tag{3.30}$$

If the flip-flop does enter metastability—that is, with probability P(samples changing input)—the time to resolve from metastability depends on the inner workings of the circuit. This resolution time determines P(unresolved), the probability that the flip-flop has not yet resolved

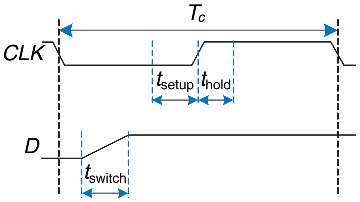


Figure 3.54 **Input timing**

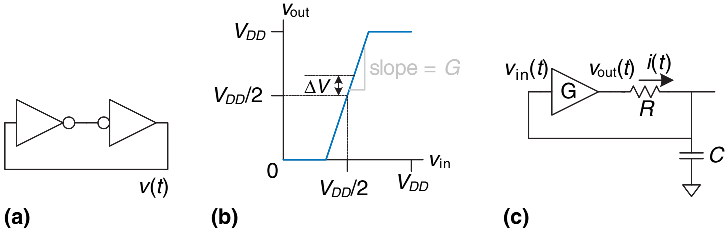


Figure 3.55 **Circuit model of bistable device**

to a valid logic level after a time t. The remainder of this section analyzes a simple model of a bistable device to estimate this probability.

A bistable device uses storage with positive feedback. Figure 3.55(a) shows this feedback implemented with a pair of inverters; this circuit's behavior is representative of most bistable elements. A pair of inverters behaves like a buffer. Let us model the buffer as having the symmetric DC transfer characteristics shown in Figure 3.55(b), with a slope of G. The buffer can deliver only a finite amount of output current; we can model this as an output resistance, R. All real circuits also have some capacitance C that must be charged up. Charging the capacitor through the resistor causes an RC delay, preventing the buffer from switching instantaneously. Hence, the complete circuit model is shown in Figure 3.55(c), where $v_{\text{out}}(t)$ is the voltage of interest conveying the state of the bistable device.

The metastable point for this circuit is $v_{\text{out}}(t) = v_{\text{in}}(t) = V_{DD}/2$; if the circuit began at exactly that point, it would remain there indefinitely in the absence of noise. Because voltages are continuous variables, the chance that the circuit will begin at exactly the metastable point is vanishingly small. However, the circuit might begin at time 0 near metastability at $v_{\text{out}}(0) = V_{DD}/2 + \Delta V$ for some small offset ΔV. In such a case, the positive feedback will eventually drive $v_{\text{out}}(t)$ to V_{DD} if $\Delta V > 0$ and to 0 if $\Delta V < 0$. The time required to reach V_{DD} or 0 is the resolution time of the bistable device.

The DC transfer characteristic is nonlinear, but it appears linear near the metastable point, which is the region of interest to us. Specifically, if $v_{\text{in}}(t) = V_{DD}/2 + \Delta V/G$, then $v_{\text{out}}(t) = V_{DD}/2 + \Delta V$ for small ΔV. The current through the resistor is $i(t) = (v_{\text{out}}(t) - v_{\text{in}}(t))/R$. The capacitor charges at a rate $dv_{\text{in}}(t)/dt = i(t)/C$. Putting these facts together, we find the governing equation for the output voltage.

$$\frac{dv_{\text{out}}(t)}{dt} = \frac{(G-1)}{RC}\left[v_{\text{out}}(t) - \frac{V_{DD}}{2}\right] \tag{3.31}$$

This is a linear first-order differential equation. Solving it with the initial condition $v_{\text{out}}(0) = V_{DD}/2 + \Delta V$ gives

$$v_{\text{out}}(t) = \frac{V_{DD}}{2} + \Delta V e^{\frac{(G-1)t}{RC}} \tag{3.32}$$

Figure 3.56 plots trajectories for $v_{\text{out}}(t)$ given various starting points. $v_{\text{out}}(t)$ moves exponentially away from the metastable point $V_{DD}/2$ until it saturates at V_{DD} or 0. The output eventually resolves to 1 or 0. The amount of time this takes depends on the initial voltage offset (ΔV) from the metastable point ($V_{DD}/2$).

Solving Equation 3.32 for the resolution time t_{res}, such that $v_{\text{out}}(t_{res}) = V_{DD}$ or 0, gives

$$|\Delta V| e^{\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2} \tag{3.33}$$

$$t_{res} = \frac{RC}{G-1} \ln \frac{V_{DD}}{2|\Delta V|} \tag{3.34}$$

In summary, the resolution time increases if the bistable device has high resistance or capacitance that causes the output to change slowly. It decreases if the bistable device has high *gain*, G. The resolution time also increases logarithmically as the circuit starts closer to the metastable point ($\Delta V \to 0$).

Define τ as $\frac{RC}{G-1}$. Solving Equation 3.34 for ΔV finds the initial offset, ΔV_{res}, that gives a particular resolution time, t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-t_{res}/\tau} \tag{3.35}$$

Suppose that the bistable device samples the input while it is changing. It measures a voltage, $v_{\text{in}}(0)$, which we will assume is uniformly distributed

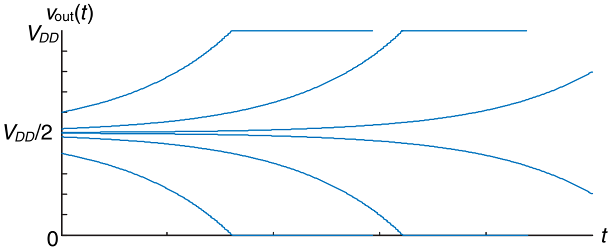


Figure 3.56 **Resolution trajectories**

between 0 and V_{DD}. The probability that the output has not resolved to a legal value after time t_{res} depends on the probability that the initial offset is sufficiently small. Specifically, the initial offset on v_{out} must be less than ΔV_{res}, so the initial offset on v_{in} must be less than $\Delta V_{res}/G$. Then the probability that the bistable device samples the input at a time to obtain a sufficiently small initial offset is

$$P(\text{unresolved}) = P\left(\left|v_{\text{in}}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{GV_{DD}} \qquad (3.36)$$

Putting this all together, the probability that the resolution time exceeds some time t is given by the following equation:

$$P(t_{res} > t) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{GT_c} e^{-\frac{t}{\tau}} \qquad (3.37)$$

Observe that Equation 3.37 is in the form of Equation 3.24, where $T_0 = (t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}})/G$ and $\tau = RC/(G-1)$. In summary, we have derived Equation 3.24 and shown how T_0 and τ depend on physical properties of the bistable device.

3.6 PARALLELISM

The speed of a system is characterized by the latency and throughput of information moving through it. We define a *token* to be a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualize data moving through the circuit. The *latency* of a system is the time required for one token to pass through the system from start to end. The *throughput* is the number of tokens that can be produced per unit time.

Example 3.15 COOKIE THROUGHPUT AND LATENCY

Ben Bitdiddle is throwing a milk and cookies party to celebrate the installation of his traffic light controller. It takes him 5 minutes to roll cookies and place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

Solution: In this example, a tray of cookies is a token. The latency is 1/3 hour per tray. The throughput is 3 trays/hour.

As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism*, and it comes in two forms: spatial and temporal. With *spatial parallelism*, multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With *temporal parallelism*, a task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap. Temporal parallelism is commonly called *pipelining*. Spatial parallelism is sometimes just called parallelism, but we will avoid that naming convention because it is ambiguous.

Example 3.16 COOKIE PARALLELISM

Ben Bitdiddle has hundreds of friends coming to his party and needs to bake cookies faster. He is considering using spatial and/or temporal parallelism.

Spatial Parallelism: Ben asks Alyssa P. Hacker to help out. She has her own cookie tray and oven.

Temporal Parallelism: Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

Solution: The latency is the time required to complete one task from start to finish. In all cases, the latency is 1/3 hour. If Ben starts with no cookies, the latency is the time needed for him to produce the first cookie tray.

The throughput is the number of cookie trays per hour. With spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes. Hence, the throughput doubles, to 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, for a throughput of 4 trays/hour. These are illustrated in Figure 3.57.

If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

Consider a task with latency L. In a system with no parallelism, the throughput is $1/L$. In a spatially parallel system with N copies of the hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken into N steps, or stages, of equal length. In such a case, the throughput is also N/L, and only one copy of the hardware is required. However, as the cookie example showed, finding N steps of equal length is often impractical. If the longest step has a latency L_1, the pipelined throughput is $1/L_1$.

Pipelining (temporal parallelism) is particularly attractive because it speeds up a circuit without duplicating the hardware. Instead, registers are placed between blocks of combinational logic to divide the logic into shorter stages that can run with a faster clock. The registers prevent a

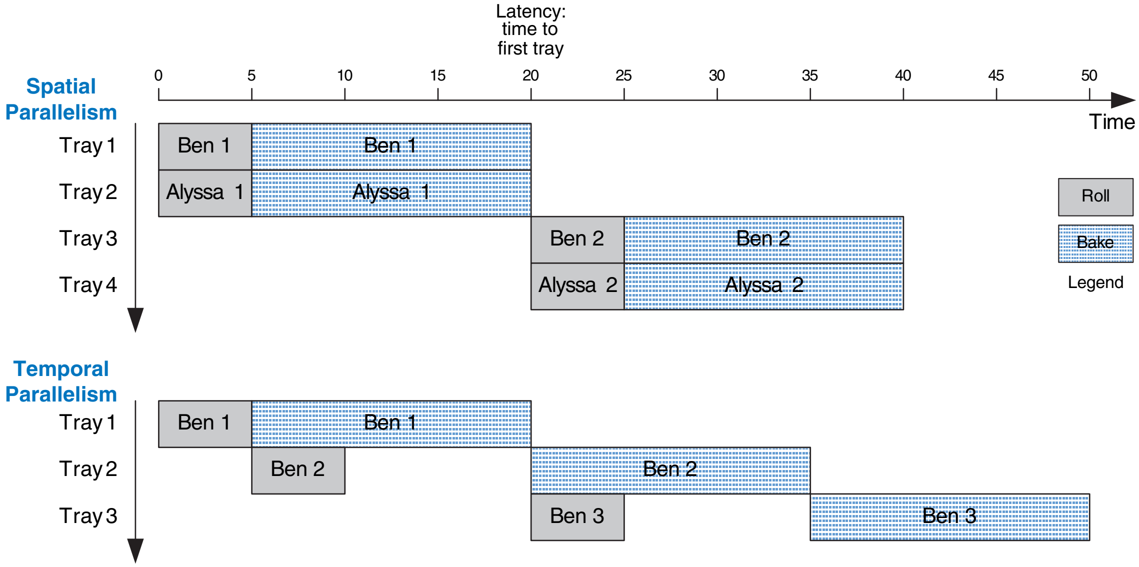


Figure 3.57 **Spatial and temporal parallelism in the cookie kitchen**

token in one pipeline stage from catching up with and corrupting the token in the next stage.

Figure 3.58 shows an example of a circuit with no pipelining. It contains four blocks of logic between the registers. The critical path passes through blocks 2, 3, and 4. Assume that the register has a clock-to-*Q* propagation delay of 0.3 ns and a setup time of 0.2 ns. Then the cycle time is $T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5$ ns. The circuit has a latency of 9.5 ns and a throughput of 1/9.5 ns = 105 MHz.

Figure 3.59 shows the same circuit partitioned into a two-stage pipeline by adding a register between blocks 3 and 4. The first stage has a minimum clock period of $0.3 + 3 + 2 + 0.2 = 5.5$ ns. The second stage has a minimum clock period of $0.3 + 4 + 0.2 = 4.5$ ns. The clock must be slow enough for all stages to work. Hence, $T_c = 5.5$ ns. The latency

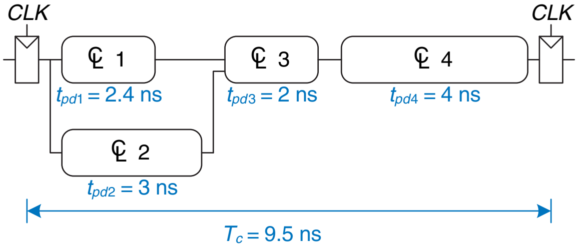


Figure 3.58 **Circuit with no pipelining**

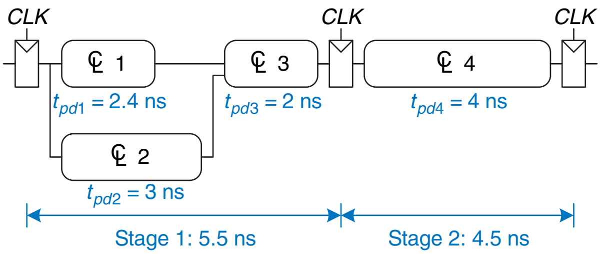

CLK
CLK
CLK
CL 1
CL 2
CL 3
CL 4
t_{pd1} = 2.4 ns
t_{pd2} = 3 ns
t_{pd3} = 2 ns
t_{pd4} = 4 ns
Stage 1: 5.5 ns
Stage 2: 4.5 ns

Figure 3.59 Circuit with two-stage pipeline

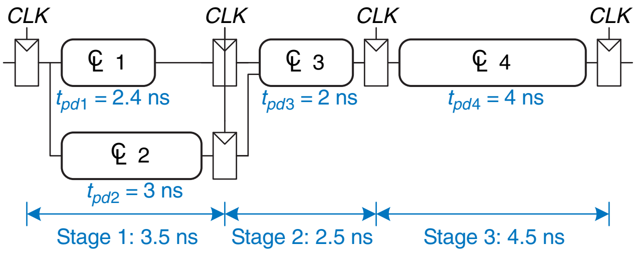

CLK
CLK
CLK
CLK
CL 1
CL 2
CL 3
CL 4
t_{pd1} = 2.4 ns
t_{pd2} = 3 ns
t_{pd3} = 2 ns
t_{pd4} = 4 ns
Stage 1: 3.5 ns
Stage 2: 2.5 ns
Stage 3: 4.5 ns

Figure 3.60 Circuit with three-stage pipeline

is two clock cycles, or 11 ns. The throughput is 1/5.5 ns = 182 MHz. This example shows that, in a real circuit, pipelining with two stages almost doubles the throughput and slightly increases the latency. In comparison, ideal pipelining would exactly double the throughput at no penalty in latency. The discrepancy comes about because the circuit cannot be divided into two exactly equal halves and because the registers introduce more sequencing overhead.

Figure 3.60 shows the same circuit partitioned into a three-stage pipeline. Note that two more registers are needed to store the results of blocks 1 and 2 at the end of the first pipeline stage. The cycle time is now limited by the third stage to 4.5 ns. The latency is three cycles, or 13.5 ns. The throughput is 1/4.5 ns = 222 MHz. Again, adding a pipeline stage improves throughput at the expense of some latency.

Although these techniques are powerful, they do not apply to all situations. The bane of parallelism is *dependencies*. If a current task is dependent on the result of a prior task, rather than just prior steps in the current task, the task cannot start until the prior task has completed. For example, if Ben wants to check that the first tray of cookies tastes good before he starts preparing the second, he has a dependency that prevents pipelining or parallel operation. Parallelism is one of the most important techniques for designing high-performance digital systems. Chapter 7 discusses pipelining further and shows examples of handling dependencies.

3.7 SUMMARY

This chapter has described the analysis and design of sequential logic. In contrast to combinational logic, whose outputs depend only on the current inputs, sequential logic outputs depend on both current and prior inputs. In other words, sequential logic remembers information about prior inputs. This memory is called the state of the logic.

Anyone who could invent logic whose outputs depend on future inputs would be fabulously wealthy!

Sequential circuits can be difficult to analyze and are easy to design incorrectly, so we limit ourselves to a small set of carefully designed building blocks. The most important element for our purposes is the flip-flop, which receives a clock and an input D and produces an output Q. The flip-flop copies D to Q on the rising edge of the clock and otherwise remembers the old state of Q. A group of flip-flops sharing a common clock is called a register. Flip-flops may also receive reset or enable control signals.

Although many forms of sequential logic exist, we discipline ourselves to use synchronous sequential circuits because they are easy to design. Synchronous sequential circuits consist of blocks of combinational logic separated by clocked registers. The state of the circuit is stored in the registers and updated only on clock edges.

Finite state machines are a powerful technique for designing sequential circuits. To design an FSM, first identify the inputs and outputs of the machine and sketch a state transition diagram, indicating the states and the transitions between them. Select an encoding for the states, and rewrite the diagram as a state transition table and output table, indicating the next state and output given the current state and input. From these tables, design the combinational logic to compute the next state and output, and sketch the circuit.

Synchronous sequential circuits have a timing specification including the clock-to-Q propagation and contamination delays, t_{pcq} and t_{ccq}, and the setup and hold times, t_{setup} and t_{hold}. For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock. The minimum cycle time T_c of the system is equal to the propagation delay t_{pd} through the combinational logic plus $t_{pcq} + t_{\text{setup}}$ of the register. For correct operation, the contamination delay through the register and combinational logic must be greater than t_{hold}. Despite the common misconception to the contrary, hold time does not affect the cycle time.

Overall system performance is measured in latency and throughput. The latency is the time required for a token to pass from start to end. The throughput is the number of tokens that the system can process per unit time. Parallelism improves system throughput.

Exercises

Exercise 3.1 Given the input waveforms shown in Figure 3.61, sketch the output, Q, of an SR latch.

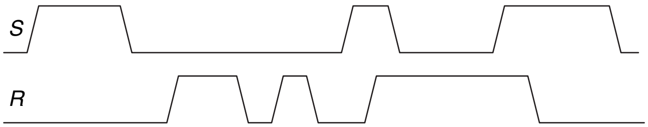


Figure 3.61 Input waveforms of SR latch for Exercise 3.1

Exercise 3.2 Given the input waveforms shown in Figure 3.62, sketch the output, Q, of an SR latch.

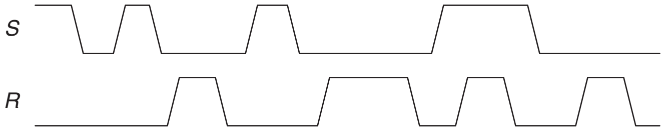


Figure 3.62 Input waveforms of SR latch for Exercise 3.2

Exercise 3.3 Given the input waveforms shown in Figure 3.63, sketch the output, Q, of a D latch.

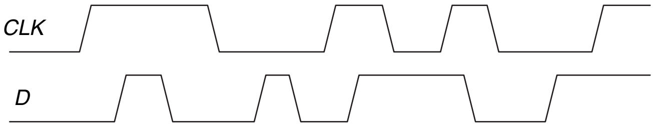


Figure 3.63 Input waveforms of D latch or flip-flop for Exercises 3.3 and 3.5

Exercise 3.4 Given the input waveforms shown in Figure 3.64, sketch the output, Q, of a D latch.

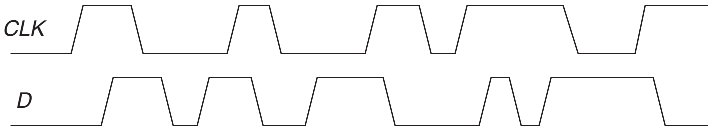


Figure 3.64 Input waveforms of D latch or flip-flop for Exercises 3.4 and 3.6

Exercise 3.5 Given the input waveforms shown in Figure 3.63, sketch the output, Q, of a D flip-flop.

Exercise 3.6 Given the input waveforms shown in Figure 3.64, sketch the output, Q, of a D flip-flop.

Exercise 3.7 Is the circuit in Figure 3.65 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

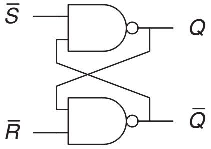


Figure 3.65 **Mystery circuit**

Exercise 3.8 Is the circuit in Figure 3.66 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

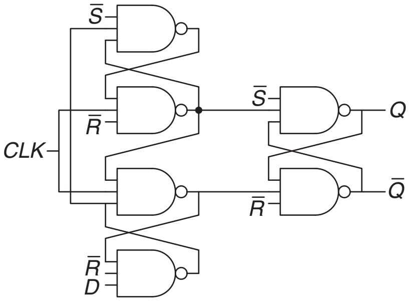


Figure 3.66 **Mystery circuit**

Exercise 3.9 The *toggle (T) flip-flop* has one input, CLK, and one output, Q. On each rising edge of CLK, Q toggles to the complement of its previous value. Draw a schematic for a T flip-flop using a D flip-flop and an inverter.

Exercise 3.10 *A JK flip-flop* receives a clock and two inputs, J and K. On the rising edge of the clock, it updates the output, Q. If J and K are both 0, Q retains its old value. If only J is 1, Q becomes 1. If only K is 1, Q becomes 0. If both J and K are 1, Q becomes the opposite of its present state.

(a) Construct a JK flip-flop using a D flip-flop and some combinational logic.

(b) Construct a D flip-flop using a JK flip-flop and some combinational logic.

(c) Construct a T flip-flop (see Exercise 3.9) using a JK flip-flop.

Exercise 3.11 The circuit in Figure 3.67 is called a *Muller C-element*. Explain in a simple fashion what the relationship is between the inputs and output.

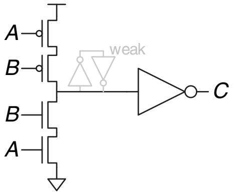


Figure 3.67 Muller C-element

Exercise 3.12 Design an asynchronously resettable D latch using logic gates.

Exercise 3.13 Design an asynchronously resettable D flip-flop using logic gates.

Exercise 3.14 Design a synchronously settable D flip-flop using logic gates.

Exercise 3.15 Design an asynchronously settable D flip-flop using logic gates.

Exercise 3.16 Suppose a ring oscillator is built from N inverters connected in a loop. Each inverter has a minimum delay of t_{cd} and a maximum delay of t_{pd}. If N is odd, determine the range of frequencies at which the oscillator might operate.

Exercise 3.17 Why must N be odd in Exercise 3.16?

Exercise 3.18 Which of the circuits in Figure 3.68 are synchronous sequential circuits? Explain.

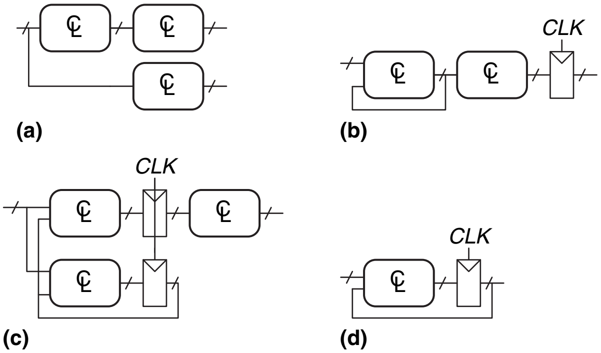


Figure 3.68 Circuits

Exercise 3.19 You are designing an elevator controller for a building with 25 floors. The controller has two inputs: *UP* and *DOWN*. It produces an output indicating the floor that the elevator is on. There is no floor 13. What is the minimum number of bits of state in the controller?

Exercise 3.20 You are designing an FSM to keep track of the mood of four students working in the digital design lab. Each student's mood is either HAPPY (the circuit works), SAD (the circuit blew up), BUSY (working on the circuit), CLUELESS (confused about the circuit), or ASLEEP (face down on the circuit board). How many states does the FSM have? What is the minimum number of bits necessary to represent these states?

Exercise 3.21 How would you factor the FSM from Exercise 3.20 into multiple simpler machines? How many states does each simpler machine have? What is the minimum total number of bits necessary in this factored design?

Exercise 3.22 Describe in words what the state machine in Figure 3.69 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

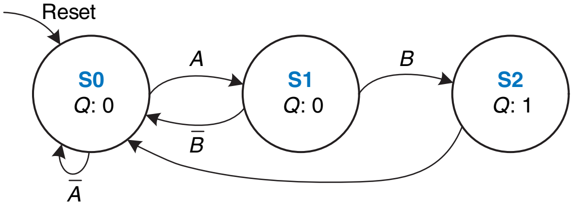


Figure 3.69 State transition diagram

Exercise 3.23 Describe in words what the state machine in Figure 3.70 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

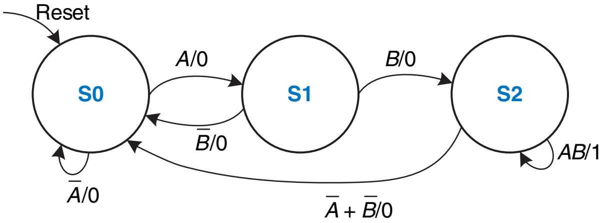


Figure 3.70 State transition diagram

Exercise 3.24 Accidents are still occurring at the intersection of Academic Avenue and Bravado Boulevard. The football team is rushing into the intersection the moment light *B* turns green. They are colliding with sleep-deprived CS majors who stagger into the intersection just before light *A* turns red. Extend the traffic light controller from Section 3.4.1 so that both lights are red for 5 seconds before either light turns green again. Sketch your improved Moore machine state transition diagram, state encodings, state transition table, output table, next state and output equations, and your FSM schematic.

Exercise 3.25 Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 or the pattern 1110. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a combined state transition and output table using your encodings. Write the next state and output equations and sketch your FSM schematic.

Exercise 3.26 You have been enlisted to design a soda machine dispenser for your department lounge. Sodas are partially subsidized by the student chapter of the IEEE, so they cost only 25 cents. The machine accepts nickels, dimes, and quarters. When enough coins have been inserted, it dispenses the soda and returns any necessary change. Design an FSM controller for the soda machine. The FSM inputs are *Nickel*, *Dime*, and *Quarter*, indicating which coin was inserted. Assume that exactly one coin is inserted on each cycle. The outputs are *Dispense*, *ReturnNickel*, *ReturnDime*, and *ReturnTwoDimes*. When the FSM reaches 25 cents, it asserts *Dispense* and the necessary *Return* outputs required to deliver the appropriate change. Then it should be ready to start accepting coins for another soda.

Exercise 3.27 Gray codes have a useful property in that consecutive numbers differ in only a single bit position. Table 3.23 lists a 3-bit Gray code representing the numbers 0 to 7. Design a 3-bit modulo 8 Gray code counter FSM with no inputs and three outputs. (A modulo N counter counts from 0 to $N - 1$, then

Table 3.23 **3-bit Gray code**

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

repeats. For example, a watch uses a modulo 60 counter for the minutes and seconds that counts from 0 to 59.) When reset, the output should be 000. On each clock edge, the output should advance to the next Gray code. After reaching 100, it should repeat with 000.

Exercise 3.28 Extend your modulo 8 Gray code counter from Exercise 3.27 to be an UP/DOWN counter by adding an *UP* input. If $UP = 1$, the counter advances to the next number. If $UP = 0$, the counter retreats to the previous number.

Exercise 3.29 Your company, Detect-o-rama, would like to design an FSM that takes two inputs, *A* and *B*, and generates one output, *Z*. The output in cycle n, Z_n, is either the Boolean AND or OR of the corresponding input A_n and the previous input $A_{n\text{-}1}$, depending on the other input, B_n:

$$\begin{aligned} Z_n &= A_n A_{n-1} && \text{if } B_n = 0 \\ Z_n &= A_n + A_{n-1} && \text{if } B_n = 1 \end{aligned}$$

(a) Sketch the waveform for *Z* given the inputs shown in Figure 3.71.

(b) Is this FSM a Moore or a Mealy machine?

(c) Design the FSM. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

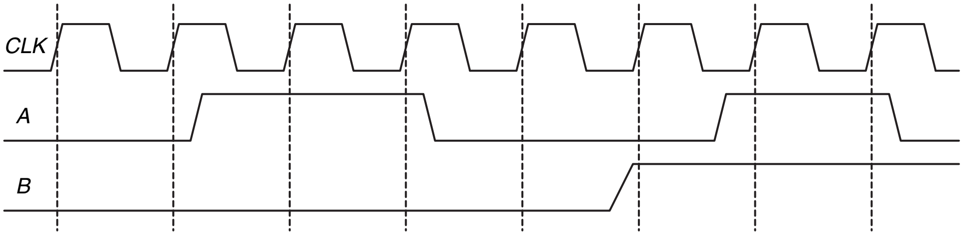


Figure 3.71 FSM input waveforms

Exercise 3.30 Design an FSM with one input, *A*, and two outputs, *X* and *Y*. *X* should be 1 if *A* has been 1 for at least three cycles altogether (not necessarily consecutively). *Y* should be 1 if *A* has been 1 for at least two consecutive cycles. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

Exercise 3.31 Analyze the FSM shown in Figure 3.72. Write the state transition and output tables and sketch the state transition diagram. Describe in words what the FSM does.

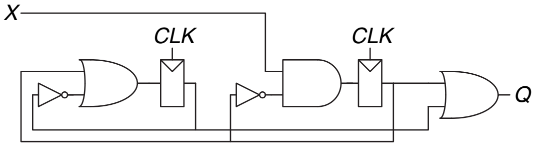


Figure 3.72 **FSM schematic**

Exercise 3.32 Repeat Exercise 3.31 for the FSM shown in Figure 3.73. Recall that the *s* and *r* register inputs indicate set and reset, respectively.

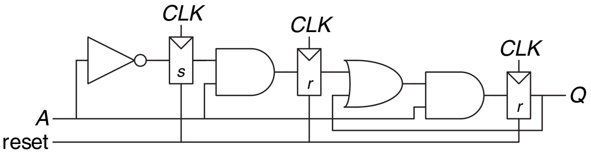


Figure 3.73 **FSM schematic**

Exercise 3.33 Ben Bitdiddle has designed the circuit in Figure 3.74 to compute a registered four-input XOR function. Each two-input XOR gate has a propagation delay of 100 ps and a contamination delay of 55 ps. Each flip-flop has a setup time of 60 ps, a hold time of 20 ps, a clock-to-*Q* maximum delay of 70 ps, and a clock-to-*Q* minimum delay of 50 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 2 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

(d) Alyssa P. Hacker points out that she can redesign the combinational logic between the registers to be faster *and* tolerate more clock skew. Her improved circuit also uses three two-input XORs, but they are arranged differently. What is her circuit? What is its maximum frequency if there is no clock skew? How much clock skew can the circuit tolerate before it might experience a hold time violation?

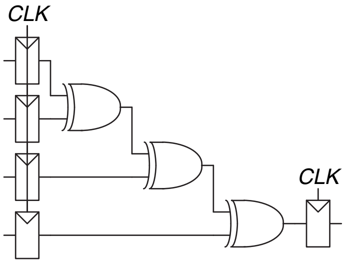


Figure 3.74 **Registered four-input XOR circuit**

Exercise 3.34 You are designing an adder for the blindingly fast 2-bit RePentium Processor. The adder is built from two full adders such that the carry out of the first adder is the carry in to the second adder, as shown in Figure 3.75. Your adder has input and output registers and must complete the addition in one clock cycle. Each full adder has the following propagation delays: 20 ps from C_{in} to C_{out} or to *Sum* (*S*), 25 ps from *A* or *B* to C_{out}, and 30 ps from *A* or *B* to *S*. The adder has a contamination delay of 15 ps from C_{in} to either output and 22 ps from *A* or *B* to either output. Each flip-flop has a setup time of 30 ps, a hold time of 10 ps, a clock-to-*Q* propagation delay of 35 ps, and a clock-to-*Q* contamination delay of 21 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 8 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

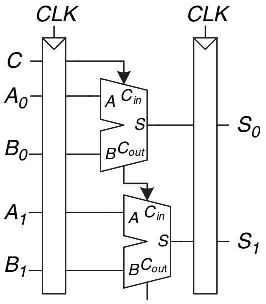


Figure 3.75 **2-bit adder schematic**

Exercise 3.35 A *field programmable gate array* (*FPGA*) uses *configurable logic blocks* (*CLBs*) rather than logic gates to implement combinational logic. The Xilinx Spartan 3 FPGA has propagation and contamination delays of 0.61 and 0.30 ns, respectively, for each CLB. It also contains flip-flops with propagation and contamination delays of 0.72 and 0.50 ns, and setup and hold times of 0.53 and 0 ns, respectively.

(a) If you are building a system that needs to run at 40 MHz, how many consecutive CLBs can you use between two flip-flops? Assume there is no clock skew and no delay through wires between CLBs.

(b) Suppose that all paths between flip-flops pass through at least one CLB. How much clock skew can the FPGA have without violating the hold time?

Exercise 3.36 A synchronizer is built from a pair of flip-flops with $t_{\text{setup}} = 50$ ps, $T_0 = 20$ ps, and $\tau = 30$ ps. It samples an asynchronous input that changes 10^8 times per second. What is the minimum clock period of the synchronizer to achieve a mean time between failures (MTBF) of 100 years?

Exercise 3.37 You would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 50 years. Your system is running at 1 GHz, and you use sampling flip-flops with $\tau = 100$ ps, $T_0 = 110$ ps, and $t_{setup} = 70$ ps. The synchronizer receives a new asynchronous input on average 0.5 times per second (i.e., once every 2 seconds). What is the required probability of failure to satisfy this MTBF? How many clock cycles would you have to wait before reading the sampled input signal to give that probability of error?

Exercise 3.38 You are walking down the hallway when you run into your lab partner walking in the other direction. The two of you first step one way and are still in each other's way. Then you both step the other way and are still in each other's way. Then you both wait a bit, hoping the other person will step aside. You can model this situation as a metastable point and apply the same theory that has been applied to synchronizers and flip-flops. Suppose you create a mathematical model for yourself and your lab partner. You start the unfortunate encounter in the metastable state. The probability that you remain in this state after t seconds is $e^{-\frac{t}{\tau}}$. τ indicates your response rate; today, your brain has been blurred by lack of sleep and has $\tau = 20$ seconds.

(a) How long will it be until you have 99% certainty that you will have resolved from metastability (i.e., figured out how to pass one another)?

(b) You are not only sleepy, but also ravenously hungry. In fact, you will starve to death if you don't get going to the cafeteria within 3 minutes. What is the probability that your lab partner will have to drag you to the morgue?

Exercise 3.39 You have built a synchronizer using flip-flops with $T_0 = 20$ ps and $\tau = 30$ ps. Your boss tells you that you need to increase the MTBF by a factor of 10. By how much do you need to increase the clock period?

Exercise 3.40 Ben Bitdiddle invents a new and improved synchronizer in Figure 3.76 that he claims eliminates metastability in a single cycle. He explains that the circuit in box M is an analog "metastability detector" that produces a HIGH output if the input voltage is in the forbidden zone between V_{IL} and V_{IH}. The metastability detector checks to determine whether the first flip-flop has produced a metastable output on $D2$. If so, it asynchronously resets the flip-flop to produce a good 0 at $D2$. The second flip-flop then samples $D2$, always producing a valid logic level on Q. Alyssa P. Hacker tells Ben that there must be a bug in the circuit, because eliminating metastability is just as impossible as building a perpetual motion machine. Who is right? Explain, showing Ben's error or showing why Alyssa is wrong.

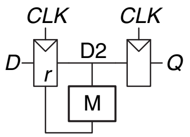


Figure 3.76 **"New and improved" synchronizer**

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 3.1 Draw a state machine that can detect when it has received the serial input sequence 01010.

Question 3.2 Design a serial (one bit at a time) two's complementer FSM with two inputs, *Start* and *A*, and one output, *Q*. A binary number of arbitrary length is provided to input *A*, starting with the least significant bit. The corresponding bit of the output appears at *Q* on the same cycle. *Start* is asserted for one cycle to initialize the FSM before the least significant bit is provided.

Question 3.3 What is the difference between a latch and a flip-flop? Under what circumstances is each one preferable?

Question 3.4 Design a 5-bit counter finite state machine.

Question 3.5 Design an edge detector circuit. The output should go HIGH for one cycle after the input makes a 0 → 1 transition.

Question 3.6 Describe the concept of pipelining and why it is used.

Question 3.7 Describe what it means for a flip-flop to have a negative hold time.

Question 3.8 Given signal *A*, shown in Figure 3.77, design a circuit that produces signal *B*.

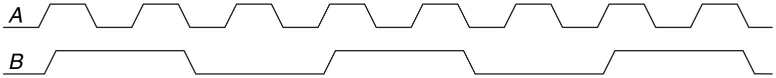


Figure 3.77 Signal waveforms

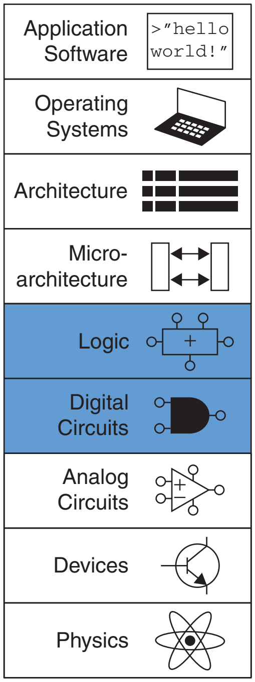
Question 3.9 Consider a block of logic between two registers. Explain the timing constraints. If you add a buffer on the clock input of the receiver (the second flip-flop), does the setup time constraint get better or worse?

FSM
HDL Dictionary
verilog
VHDL
a & b
a and b
a ^ b
a xor b
~(a+b)
a nor b
Hardware

Hardware Description Languages

4

4.1 Introduction
4.2 Combinational Logic
4.3 Structural Modeling
4.4 Sequential Logic
4.5 More Combinational Logic
4.6 Finite State Machines
4.7 Data Types*
4.8 Parameterized Modules*
4.9 Testbenches
4.10 Summary
Exercises
Interview Questions



4.1 INTRODUCTION

Thus far, we have focused on designing combinational and sequential digital circuits at the schematic level. The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations and manual translation of finite state machines (FSMs) into gates. In the 1990s, designers discovered that they were far more productive if they worked at a higher level of abstraction, specifying just the logical function and allowing a *computer-aided design* (*CAD*) tool to produce the optimized gates. The specifications are generally given in a *hardware description language* (*HDL*). The two leading hardware description languages are *SystemVerilog* and *VHDL*.

SystemVerilog and VHDL are built on similar principles but have different syntax. Discussion of these languages in this chapter is divided into two columns for literal side-by-side comparison, with SystemVerilog on the left and VHDL on the right. When you read the chapter for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it.

Subsequent chapters show hardware in both schematic and HDL form. If you choose to skip this chapter and not learn one of the HDLs, you will still be able to master the principles of computer organization from the schematics. However, the vast majority of commercial systems are now built using HDLs rather than schematics. If you expect to do digital design at any point in your professional life, we urge you to learn one of the HDLs.

4.1.1 Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00004-5
© 2013 Elsevier, Inc. All rights reserved.

behavioral and *structural*. Behavioral models describe what a module does. Structural models describe how a module is built from simpler pieces; it is an application of hierarchy. The SystemVerilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6, $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. In both languages, the module is named `sillyfunction` and has three inputs, `a`, `b`, and `c`, and one output, `y`.

HDL Example 4.1 COMBINATIONAL LOGIC

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);
end;
```

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library use` clause will be discussed in Section 4.7.2. The `entity` declaration lists the module name and its inputs and outputs. The `architecture` body defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of `'0'` or `'1'`, as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations between AND and OR, so Boolean equations should be parenthesized.

A module, as you might expect, is a good application of modularity. It has a well defined interface, consisting of its inputs and outputs, and it performs a specific function. The particular way in which it is coded is unimportant to others that might use the module, as long as it performs its function.

4.1.2 Language Origins

Universities are almost evenly split on which of these languages is taught in a first course. Industry is trending toward SystemVerilog, but many companies still use VHDL and many designers need to be fluent in both.

SystemVerilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard[1] in 1995. The language was extended in 2005 to streamline idiosyncrasies and to better support modeling and verification of systems. These extensions have been merged into a single language standard, which is now called SystemVerilog (IEEE STD 1800-2009). SystemVerilog file names normally end in `.sv`.

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense.

VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis. The IEEE standardized it in 1987 and has updated the standard several times since. This chapter is based on the 2008 revision of the VHDL standard (IEEE STD 1076-2008), which streamlines the language in a variety of ways. At the time of this writing, not all of the VHDL 2008 features are supported by CAD tools; this chapter only uses those understood by Synplicity, Altera Quartus, and ModelSim. VHDL file names normally end in `.vhd`.

To use VHDL 2008 in ModelSim, you may need to set VHDL93 = 2008 in the modelsim.ini configuration file.

Compared to SystemVerilog, VHDL is more verbose and cumbersome, as you might expect of a language developed by committee.

Both languages are fully capable of describing any hardware system, and both have their quirks. The best language to use is the one that is already being used at your site or the one that your customers demand. Most CAD tools today allow the two languages to be mixed, so that different modules can be described in different languages.

4.1.3 Simulation and Synthesis

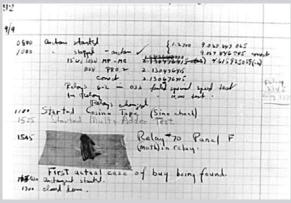
The two major purposes of HDLs are logic *simulation* and *synthesis*. During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Testing a system in the laboratory is time-consuming. Discovering the cause of errors in the lab can be extremely difficult, because only signals routed to the chip pins can be observed. There is no way to directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example,

The term "bug" predates the invention of the computer. Thomas Edison called the "little faults and difficulties" with his inventions "bugs" in 1878.

The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electromechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment "first actual case of bug being found."



Source: Notebook entry courtesy Naval Historical Center, US Navy; photo No. NII 96566-KN)

---

[1] The Institute of Electrical and Electronics Engineers (IEEE) is a professional society responsible for many computing standards including Wi-Fi (802.11), Ethernet (802.3), and floating-point numbers (754).

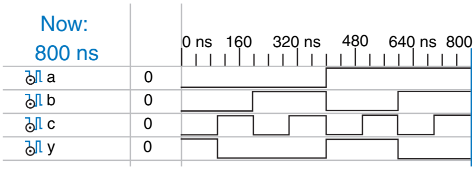


Figure 4.1 **Simulation waveforms**

correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of $475 million. Logic simulation is essential to test a system before it is built.

Figure 4.1 shows waveforms from a simulation[2] of the previous `sillyfunction` module demonstrating that the module works correctly. `y` is TRUE when `a`, `b`, and `c` are 000, 100, or 101, as specified by the Boolean equation.

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit. Figure 4.2 shows the results of synthesizing the `sillyfunction` module.[3] Notice how the three three-input AND gates are simplified into two two-input AND gates, as we discovered in Example 2.6 using Boolean algebra.

The synthesis tool labels each of the synthesized gates. In Figure 4.2, they are un5_y, un8_y, and y.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. SystemVerilog and VHDL are rich languages with many commands. Not all of these commands can be synthesized into hardware.

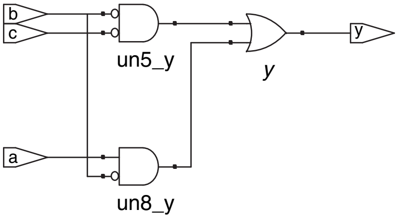


Figure 4.2 **Synthesized circuit**

---

[2] The simulation was performed with the ModelSim PE Student Edition Version 10.3c. ModelSim was selected because it is used commercially, yet a student version with a capacity of 10,000 lines of code is freely available.

[3] Synthesis was performed with Synplify Premier from Synplicity. The tool was selected because it is the leading commercial tool for synthesizing HDL to field-programmable gate arrays (see Section 5.6.2) and because it is available inexpensively for universities.

For example, a command to print results on the screen during simulation does not translate into hardware. Because our primary interest is to build hardware, we will emphasize a *synthesizable subset* of the languages. Specifically, we will divide HDL code into *synthesizable* modules and a *testbench*. The synthesizable modules describe the hardware. The testbench contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies between expected and actual outputs. Testbench code is intended only for simulation and cannot be synthesized.

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

In our experience, the best way to learn an HDL is by example. HDLs have specific ways of describing various classes of logic; these ways are called *idioms*. This chapter will teach you how to write the proper HDL idioms for each type of block and then how to put the blocks together to produce a working system. When you need to describe a particular kind of hardware, look for a similar example and adapt it to your purpose. We do not attempt to rigorously define all the syntax of the HDLs, because that is deathly boring and because it tends to encourage thinking of HDLs as programming languages, not shorthand for hardware. The IEEE SystemVerilog and VHDL specifications, and numerous dry but exhaustive textbooks, contain all of the details, should you find yourself needing more information on a particular topic. (See the Further Readings section at the back of the book.)

4.2 COMBINATIONAL LOGIC

Recall that we are disciplining ourselves to design synchronous sequential circuits, which consist of combinational logic and registers. The outputs of combinational logic depend only on the current inputs. This section describes how to write behavioral models of combinational logic with HDLs.

4.2.1 Bitwise Operators

Bitwise operators act on single-bit signals or on multi-bit busses. For example, the `inv` module in HDL Example 4.2 describes four inverters connected to 4-bit busses.

HDL Example 4.2 INVERTERS

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses `STD_LOGIC_VECTOR` to indicate busses of `STD_LOGIC`. `STD_LOGIC_VECTOR(3 downto 0)` represents a 4-bit bus. The bits, from most significant to least significant, are `a(3)`, `a(2)`, `a(1)`, and `a(0)`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be `STD_LOGIC_VECTOR(4 downto 1)`, in which case bit 4 would have been the most significant. Or we could have written `STD_LOGIC_VECTOR(0 to 3)`, in which case the bits, from most significant to least significant, would be `a(0)`, `a(1)`, `a(2)`, and `a(3)`. This is called *big-endian* order.

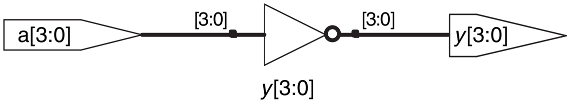


Figure 4.3 `inv` **synthesized circuit**

The endianness of a bus is purely arbitrary. (See the sidebar in Section 6.2.2 for the origin of the term.) Indeed, endianness is also irrelevant to this example, because a bank of inverters doesn't care what the order of the bits are. Endianness matters only for operators, such as addition, where the sum of one column carries over into the next. Either ordering is acceptable, as long as it is used consistently. We will consistently use the little-endian order, `[N – 1:0]` in SystemVerilog and `(N – 1 downto 0)` in VHDL, for an *N*-bit bus.

After each code example in this chapter is a schematic produced from the SystemVerilog code by the Synplify Premier synthesis tool. Figure 4.3 shows that the `inv` module synthesizes to a bank of four inverters, indicated by the inverter symbol labeled `y[3:0]`. The bank of inverters connects to 4-bit input and output busses. Similar hardware is produced from the synthesized VHDL code.

The `gates` module in HDL Example 4.3 demonstrates bitwise operations acting on 4-bit busses for other basic logic functions.

HDL Example 4.3 LOGIC GATES

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;    // AND
  assign y2 = a | b;    // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```

`~`, `^`, and `|` are examples of SystemVerilog *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a & b`, or `~(a | b)`, is called an *expression*. A complete command such as `assign y4 = ~(a & b);` is called a *statement*.

`assign out = in1 op in2;` is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- five different two-input logic gates
  -- acting on 4-bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

`not`, `xor`, and `or` are examples of VHDL *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a and b`, or `a nor b`, is called an *expression*. A complete command such as `y4 <= a nand b;` is called a *statement*.

`out <= in1 op in2;` is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Anytime the inputs on the right side of the `<=` in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.

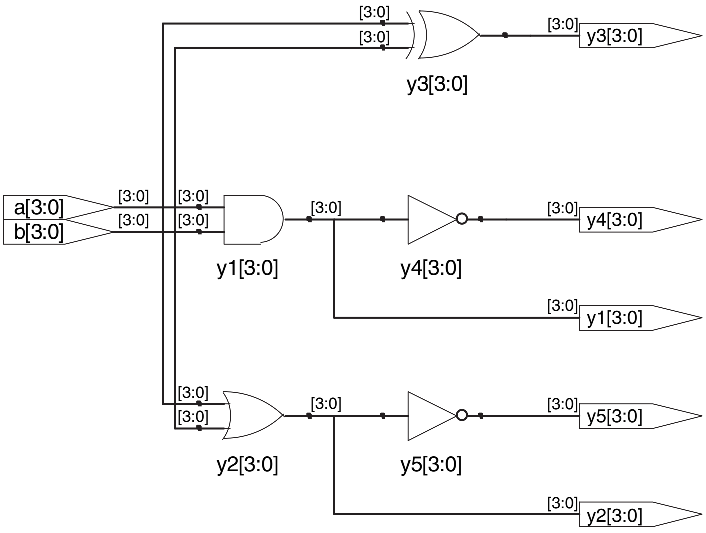


Figure 4.4 `gates` **synthesized circuit**

4.2.2 Comments and White Space

The `gates` example showed how to format comments. SystemVerilog and VHDL are not picky about the use of white space (i.e., spaces, tabs, and line breaks). Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable. Be consistent in your use of capitalization and underscores in signal and module names. This text uses all lower case. Module and signal names must not begin with a digit.

SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line.

SystemVerilog is case-sensitive. `y1` and `Y1` are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

VHDL

Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `--` continue to the end of the line.

VHDL is not case-sensitive. `y1` and `Y1` are the same signal in VHDL. However, other tools that may read your file might be case sensitive, leading to nasty bugs if you blithely mix upper and lower case.

4.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. HDL Example 4.4 describes an eight-input AND gate with inputs a_7, a_6, . . ., a_0. Analogous reduction operators exist for OR, XOR, NAND, NOR, and XNOR gates. Recall that a multiple-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

HDL Example 4.4 EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --      a(3) and a(2) and a(1) and a(0);
end;
```

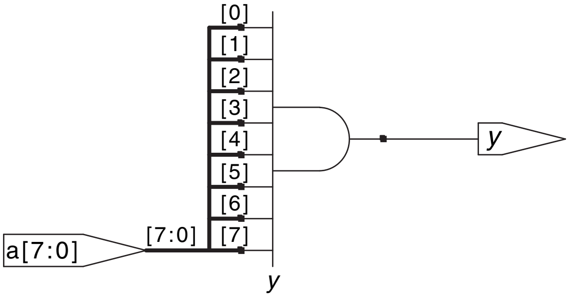


Figure 4.5 `and8` **synthesized circuit**

4.2.4 Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the *condition*. HDL Example 4.5 illustrates a 2:1 multiplexer using conditional assignment.

HDL Example 4.5 2:1 MULTIPLEXER

SystemVerilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y = d1`. If `s` is 0, then `y = d0`.

`?:` is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The conditional signal assignment sets `y` to `d1` if `s` is 1. Otherwise it sets `y` to `d0`. Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.

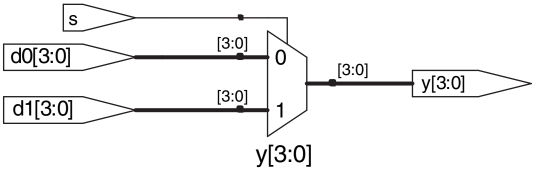


Figure 4.6 `mux2` **synthesized circuit**

HDL Example 4.6 shows a 4:1 multiplexer based on the same principle as the 2:1 multiplexer in HDL Example 4.5. Figure 4.7 shows the schematic for the 4:1 multiplexer produced by Synplify Premier. The software uses a different multiplexer symbol than this text has shown so far. The multiplexer has multiple data `(d)` and one-hot enable `(e)` inputs. When one of the enables is asserted, the associated data is passed to the output. For example, when `s[1] = s[0] = 0`, the bottom AND gate, `un1_s_5`, produces a 1, enabling the bottom input of the multiplexer and causing it to select `d0[3:0]`.

HDL Example 4.6 4:1 MULTIPLEXER

SystemVerilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` ($y = d3$ if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

VHDL

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a `switch/case` statement in place of multiple `if/else` statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

4.2.5 Internal Variables

Often it is convenient to break a complex function into intermediate steps. For example, a full adder, which will be described in Section 5.2.1,

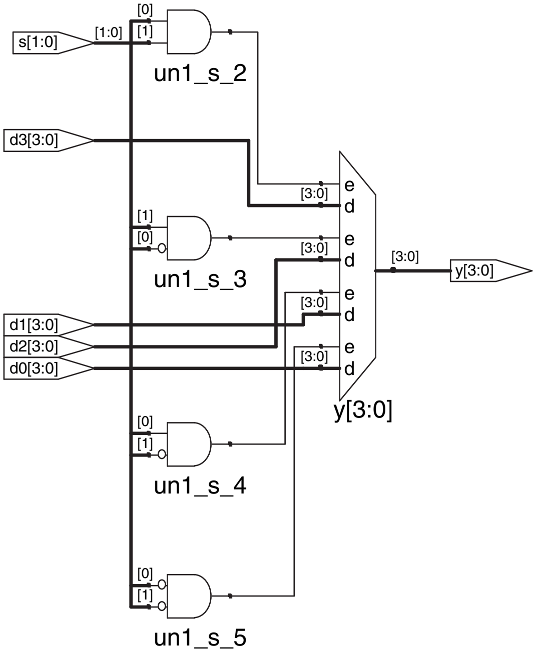


Figure 4.7 `mux4` **synthesized circuit**

is a circuit with three inputs and two outputs defined by the following equations:

$$S = A \oplus B \oplus C_{\text{in}}$$
$$C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}} \qquad (4.1)$$

If we define intermediate signals, P and G,

$$P = A \oplus B$$
$$G = AB \qquad (4.2)$$

we can rewrite the full adder as follows:

$$S = P \oplus C_{\text{in}}$$
$$C_{\text{out}} = G + PC_{\text{in}} \qquad (4.3)$$

Check this by filling out the truth table to convince yourself it is correct.

P and G are called *internal variables*, because they are neither inputs nor outputs but are used only internal to the module. They are similar to local variables in programming languages. HDL Example 4.7 shows how they are used in HDLs.

HDL assignment statements (`assign` in SystemVerilog and <= in VHDL) take place concurrently. This is different from conventional programming languages such as C or Java, in which statements are evaluated in the order in which they are written. In a conventional language, it is

HDL Example 4.7 FULL ADDER

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b`;

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

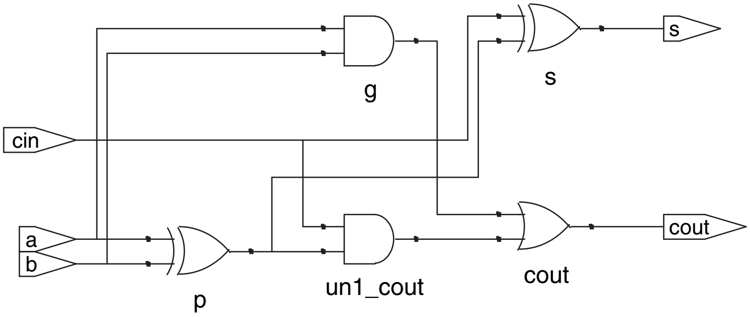


Figure 4.8 `fulladder` **synthesized circuit**

important that $S = P \oplus C_{in}$ comes after $P = A \oplus B$, because statements are executed sequentially. In an HDL, the order does not matter. Like hardware, HDL assignment statements are evaluated any time the inputs, signals on the right hand side, change their value, regardless of the order in which the assignment statements appear in a module.

4.2.6 Precedence

Notice that we parenthesized the `cout` computation in HDL Example 4.7 to define the order of operations as $C_{out} = G + (P \cdot C_{in})$, rather than $C_{out} = (G + P) \cdot C_{in}$. If we had not used parentheses, the default operation

HDL Example 4.8 OPERATOR PRECEDENCE

SystemVerilog

Table 4.1 SystemVerilog operator precedence

	Op	Meaning
Highest	`~`	NOT
	`*, /, %`	MUL, DIV, MOD
	`+, -`	PLUS, MINUS
	`<<, >>`	Logical Left/Right Shift
	`<<<, >>>`	Arithmetic Left/Right Shift
	`<, <=, >, >=`	Relative Comparison
	`==, !=`	Equality Comparison
Lowest	`&, ~&`	AND, NAND
	`^, ~^`	XOR, XNOR
	`|, ~|`	OR, NOR
	`?:`	Conditional

The operator precedence for SystemVerilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g | p & cin;
```

VHDL

Table 4.2 VHDL operator precedence

	Op	Meaning
Highest	`not`	NOT
	`*, /, mod, rem`	MUL, DIV, MOD, REM
	`+, -`	PLUS, MINUS
	`rol, ror,` `srl, sll`	Rotate, Shift logical
Lowest	`<, <=, >, >=`	Relative Comparison
	`=, /=`	Equality Comparison

	`and, or,`
	`nand, nor,`
	`xor, xnor`
		Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike SystemVerilog, all of the logical operations (`and`, `or`, etc.) have equal precedence, unlike what one might expect in Boolean algebra. Thus, parentheses are necessary; otherwise `cout <= g or p and cin` would be interpreted from left to right as `cout <= (g or p) and cin`.

order is defined by the language. HDL Example 4.8 specifies operator precedence from highest to lowest for each language. The tables include arithmetic, shift, and comparison operators that will be defined in Chapter 5.

4.2.7 Numbers

Numbers can be specified in binary, octal, decimal, or hexadecimal (bases 2, 8, 10, and 16, respectively). The size, i.e., the number of bits, may optionally be given, and leading zeros are inserted to reach this size. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. HDL Example 4.9 explains how numbers are written in each language.

HDL Example 4.9 NUMBERS

SystemVerilog

The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal. If the base is omitted, it defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Table 4.3 **SystemVerilog numbers**

Numbers	Bits	Base	Val	Stored
`3'b101`	3	2	5	101
`'b11`	?	2	3	000 … 0011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	00 … 0101010

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes: `'0'` and `'1'` indicate logic 0 and 1. The format for declaring STD_LOGIC_VECTOR constants is `NB"value"`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9X"25"` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 supports `B` for binary, `O` for octal, `D` for decimal, and `X` for hexadecimal.

If the base is omitted, it defaults to binary. If the size is not given, the number is assumed to have a size matching the number of bits specified in the value. As of October 2011, Synplify Premier from Synopsys does not yet support specifying the size.

`others => '0'` and `others => '1'` are VHDL idioms to fill all of the bits with 0 and 1, respectively.

Table 4.4 **VHDL numbers**

Numbers	Bits	Base	Val	Stored
`3B"101"`	3	2	5	101
`B"11"`	2	2	3	11
`8B"11"`	8	2	3	00000011
`8B"1010_1011"`	8	2	171	10101011
`3D"6"`	3	10	6	110
`6O"42"`	6	8	34	100010
`8X"AB"`	8	16	171	10101011
`"101"`	3	2	5	101
`B"101"`	3	2	5	101
`X"AB"`	8	16	171	10101011

4.2.8 Z's and X's

HDLs use `z` to indicate a floating value, `z` is particularly useful for describing a tristate buffer, whose output floats when the enable is 0. Recall from Section 2.6.2 that a bus can be driven by several tristate buffers, exactly one of which should be enabled. HDL Example 4.10 shows the idiom for a tristate buffer. If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (`z`).

Similarly, HDLs use `x` to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is `x`, indicating contention. If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by `z`.

At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (`x` in SystemVerilog and `u` in VHDL). This is helpful to track errors caused by forgetting to reset a flip-flop before its output is used.

HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that `y` is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also note that a `tri` output from a module can be used as a `logic` input to another module. Section 4.7 further discusses nets with multiple drivers.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= a when en else "ZZZZ";
end;
```

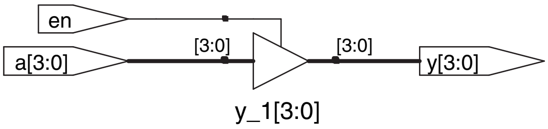


Figure 4.9 `tristate` **synthesized circuit**

If a gate receives a floating input, it may produce an `x` output when it can't determine the correct output value. Similarly, if it receives an illegal or uninitialized input, it may produce an `x` output. HDL Example 4.11

HDL Example 4.11 TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS

SystemVerilog

SystemVerilog signal values are `0`, `1`, `z`, and `x`. SystemVerilog constants starting with `z` or `x` are padded with leading `z`'s or `x`'s (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns `0` because the output of an AND gate is always `0` if either input is `0`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in SystemVerilog.

VHDL

VHDL STD_LOGIC signals are `'0'`, `'1'`, `'z'`, `'x'`, and `'u'`.

Table 4.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, `'0' and 'z'` returns `'0'` because the output of an AND gate is always `'0'` if either input is `'0'`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `'x'` in VHDL. Uninitialized inputs cause uninitialized outputs, displayed as `'u'` in VHDL.

Table 4.5 **SystemVerilog AND gate truth table with `z` and `x`**

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

Table 4.6 **VHDL AND gate truth table with `z`, `x` and `u`**

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

HDL Example 4.12 BIT SWIZZLING

SystemVerilog

```
assign y={c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The `{}` operator is used to concatenate busses. `{3{d[0]}}` indicates three copies of `d[0]`.

Don't confuse the 3-bit binary constant `3'b101` with a bus named `b`. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of `y`.

If `y` were wider than 9 bits, zeros would be placed in the most significant bits.

VHDL

```
y <=(c(2 downto 1), d(0), d(0), d(0), c(0), 3B"101");
```

The `()` aggregate operator is used to concatenate busses. `y` must be a 9-bit `STD_LOGIC_VECTOR`.

Another example demonstrates the power of VHDL aggregations. Assuming `z` is an 8-bit STD_LOGIC_VECTOR, `z` is given the value 10010110 using the following command aggregation.

```
z <= ("10", 4 => '1', 2 downto 1 =>'1', others =>'0')
```

The `"10"` goes in the leading pair of bits. 1s are also placed into bit 4 and bits 2 and 1. The other bits are 0.

---

shows how SystemVerilog and VHDL combine these different signal values in logic gates.

Seeing `x` or `u` values in simulation is almost always an indication of a bug or bad coding practice. In the synthesized circuit, this corresponds to a floating gate input, uninitialized state, or contention. The `x` or `u` may be interpreted randomly by the circuit as 0 or 1, leading to unpredictable behavior.

4.2.9 Bit Swizzling

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as *bit swizzling*. In HDL Example 4.12, `y` is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations.

4.2.10 Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and also for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results). These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13 adds delays to the original function from HDL Example 4.1, $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. It assumes that inverters have a delay of 1 ns, three-input AND gates have a delay of 2 ns, and three-input OR gates have a delay of 4 ns. Figure 4.10 shows the simulation waveforms, with `y` lagging 7 ns after the inputs. Note that `y` is initially unknown at the beginning of the simulation.

HDL Example 4.13 LOGIC GATES WITH DELAYS

SystemVerilog

```
‘timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `'timescale unit/precision`. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as non-blocking (`<=`) and blocking (`=`) assignments, which will be discussed in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

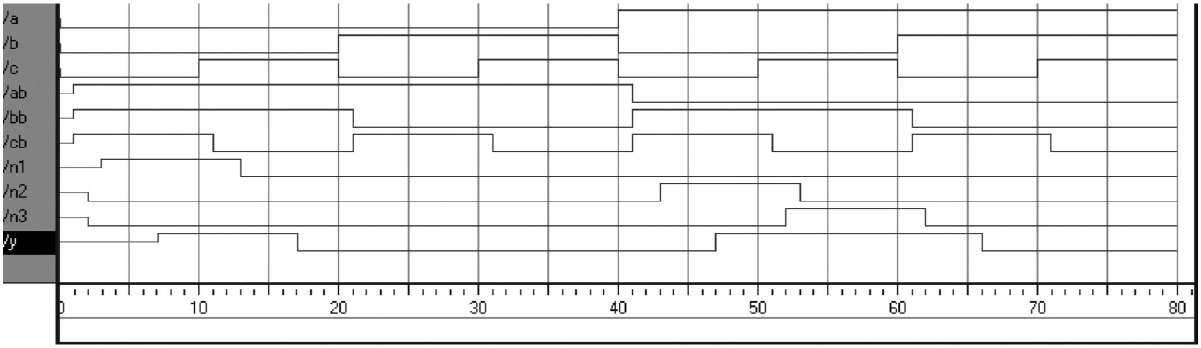


Figure 4.10 Example simulation waveforms with delays (from the ModelSim simulator)

4.3 STRUCTURAL MODELING

The previous section discussed *behavioral* modeling, describing a module in terms of the relationships between inputs and outputs. This section examines *structural* modeling, describing a module in terms of how it is composed of simpler modules.

For example, HDL Example 4.14 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers. Each copy of the 2:1 multiplexer is called

---

HDL Example 4.14 STRUCTURAL MODEL OF 4:1 MULTIPLEXER

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the SystemVerilog code — see HDL Example 4.5, 4.15, or 4.34.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the `mux2` ports using the `component` declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of `mux4` was named `struct`, whereas architectures of modules with behavioral descriptions from Section 4.2 were named `synth`. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but `struct` and `synth` are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

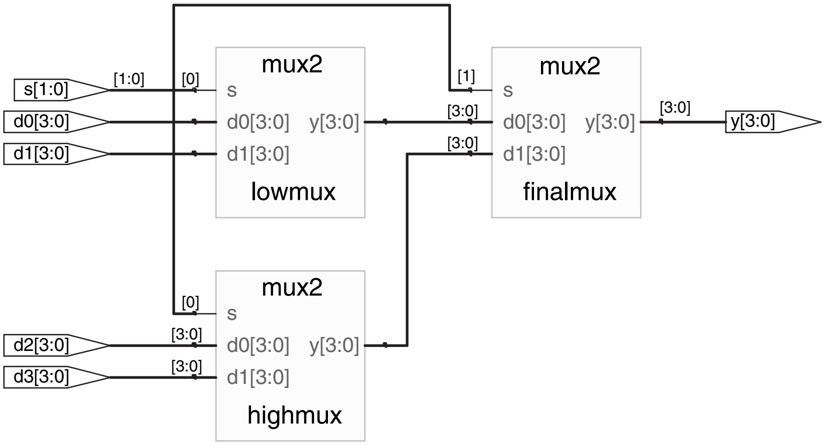


Figure 4.11 `mux4` **synthesized circuit**

an *instance*. Multiple instances of the same module are distinguished by distinct names, in this case `lowmux`, `highmux`, and `finalmux`. This is an example of regularity, in which the 2:1 multiplexer is reused many times.

HDL Example 4.15 uses structural modeling to construct a 2:1 multiplexer from a pair of tristate buffers. Building logic out of tristates is not recommended, however.

HDL Example 4.15 STRUCTURAL MODEL OF 2:1 MULTIPLEXER

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.

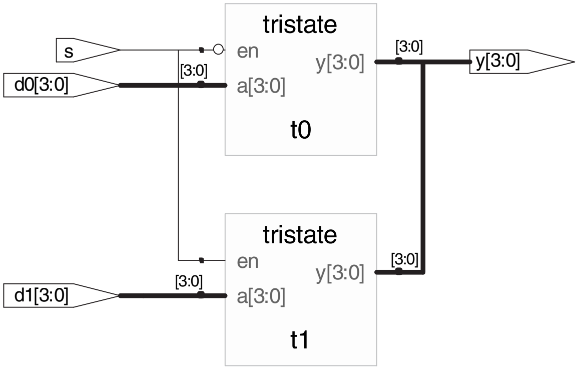


Figure 4.12 `mux2` **synthesized circuit**

HDL Example 4.16 shows how modules can access part of a bus. An 8-bit wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.

In general, complex systems are designed *hierarchically*. The overall system is described structurally by instantiating its major components. Each of these components is described structurally from its building blocks, and so forth recursively until the pieces are simple enough to describe behaviorally. It is good style to avoid (or at least to minimize) mixing structural and behavioral descriptions within a single module.

HDL Example 4.16 ACCESSING PARTS OF BUSSES

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

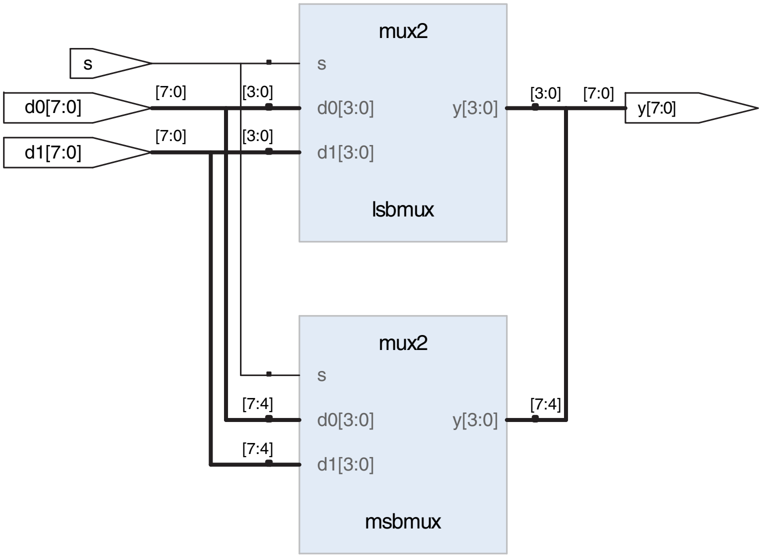


Figure 4.13 `mux2_8` **synthesized circuit**

4.4 SEQUENTIAL LOGIC

HDL synthesizers recognize certain idioms and turn them into specific sequential circuits. Other coding styles may simulate correctly but synthesize into circuits with blatant or subtle errors. This section presents the proper idioms to describe registers and latches.

4.4.1 Registers

The vast majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops. HDL Example 4.17 shows the idiom for such flip-flops.

In SystemVerilog `always` statements and VHDL `process` statements, signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory. For example, the flip-flop includes only `clk` in the sensitive list. It remembers its old value of `q` until the next rising edge of the `clk`, even if `d` changes in the interim.

In contrast, SystemVerilog continuous assignment statements (`assign`) and VHDL concurrent assignment statements (<=) are reevaluated anytime any of the inputs on the right hand side changes. Therefore, such code necessarily describes combinational logic.

HDL Example 4.17 REGISTER

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

In general, a SystemVerilog `always` statement is written in the form

```
always @(sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

<= is called a *nonblocking assignment*. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that <= is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

```
process(sensitivity list) begin
  statement;
end process;
```

The `statement` is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement checks if the change was a rising edge on clk. If so, then `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

`rising_edge(clk)` is synonymous with `clk'event and clk = '1'`.

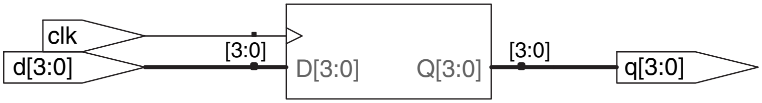


Figure 4.14 `flop` **synthesized circuit**

4.4.2 Resettable Registers

When simulation begins or power is first applied to a circuit, the output of a flop or register is unknown. This is indicated with `x` in SystemVerilog and `u` in VHDL. Generally, it is good practice to use resettable registers so that on powerup you can put your system in a known state. The reset may be either asynchronous or synchronous. Recall that asynchronous reset occurs immediately, whereas synchronous reset clears the output only on the next rising edge of the clock. HDL Example 4.18 demonstrates the idioms for flip-flops with asynchronous and synchronous resets. Note that distinguishing synchronous and asynchronous reset in a schematic can be difficult. The schematic produced by Synplify Premier places asynchronous reset at the bottom of a flip-flop and synchronous reset on the left side.

HDL Example 4.18 RESETTABLE REGISTER

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Recall that the state of a flop is initialized to `'u'` at startup during VHDL simulation.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design.

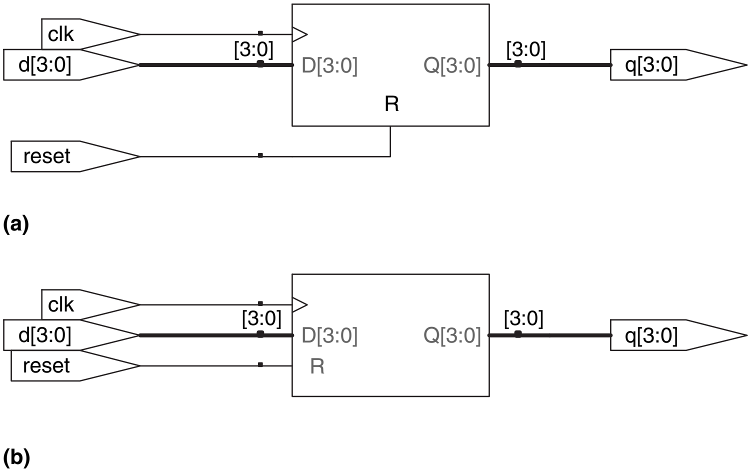


Figure 4.15 `flopr` **synthesized circuit (a) asynchronous reset, (b) synchronous reset**

4.4.3 Enabled Registers

Enabled registers respond to the clock only when the enable is asserted. HDL Example 4.19 shows an asynchronously resettable enabled register that retains its old value if both `reset` and `en` are FALSE.

HDL Example 4.19 RESETTABLE ENABLED REGISTER

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

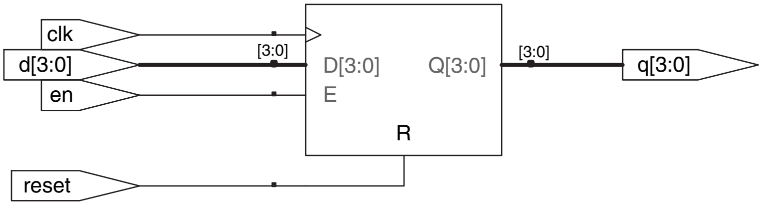


Figure 4.16 `flopenr` **synthesized circuit**

4.4.4 Multiple Registers

A single `always/process` statement can be used to describe multiple pieces of hardware. For example, consider the synchronizer from Section 3.5.5 made of two back-to-back flip-flops, as shown in Figure 4.17. HDL Example 4.20 describes the synchronizer. On the rising edge of `clk`, `d` is copied to `n1`. At the same time, `n1` is copied to `q`.

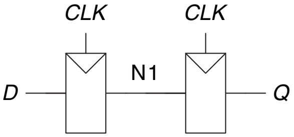


Figure 4.17 **Synchronizer circuit**

HDL Example 4.20 SYNCHRONIZER

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to `{}` in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

`n1` must be declared as a `signal` because it is an internal signal used in the module.

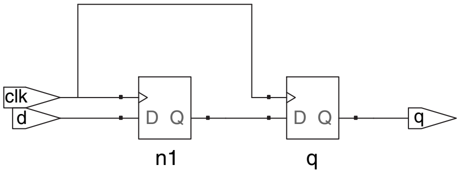


Figure 4.18 `sync` **synthesized circuit**

4.4.5 Latches

Recall from Section 3.2.2 that a D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state. HDL Example 4.21 shows the idiom for a D latch.

Not all synthesis tools support latches well. Unless you know that your tool does support latches and you have a good reason to use them, avoid them and use edge-triggered flip-flops instead. Furthermore, take care that your HDL does not imply any unintended latches, something that is easy to do if you aren't attentive. Many synthesis tools warn you when a latch is created; if you didn't expect one, track down the bug in your HDL. And if you don't know whether you intended to have a latch or not, you are probably approaching HDLs like a programming language and have bigger problems lurking.

4.5 MORE COMBINATIONAL LOGIC

In Section 4.2, we used assignment statements to describe combinational logic behaviorally. SystemVerilog `always` statements and VHDL `process`

HDL Example 4.21 D LATCH

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` is equivalent to `always @(clk, d)` and is the preferred idiom for describing a latch in SystemVerilog. It evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`, so this code describes a positive level sensitive latch. Otherwise, `q` keeps its old value. SystemVerilog can generate a warning if the `always_latch` block doesn't imply a latch.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both `clk` and `d`, so the `process` evaluates anytime `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

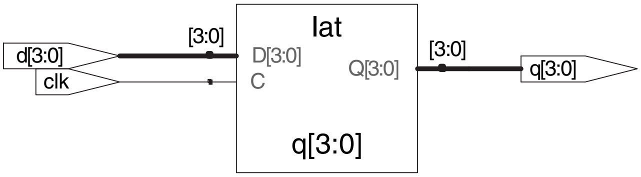


Figure 4.19 `latch` **synthesized circuit**

HDL Example 4.22 INVERTER USING `always/process`

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

`always_comb` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = in the `always` statement change. In this case, it is equivalent to `always @(a)`, but is better because it avoids mistakes if signals in the `always` statement are renamed or added. If the code inside the `always` block is not combinational logic, SystemVerilog will report a warning. `always_comb` is equivalent to `always @(*)`, but is preferred in SystemVerilog.

The = in the `always` statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In SystemVerilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

`process(all)` reevaluates the statements inside the `process` any time any of the signals in the `process` change. It is equivalent to `process(a)` but is better because it avoids mistakes if signals in the `process` are renamed or added.

The `begin` and `end process` statements are required in VHDL even though the `process` contains only one assignment.

statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed. However, `always/process` statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL Example 4.22 uses `always/process` statements to describe a bank of four inverters (see Figure 4.3 for the synthesized circuit).

HDLs support *blocking* and *nonblocking assignments* in an `always/process` statement. A group of blocking assignments are evaluated in the order in which they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left hand sides are updated.

HDL Example 4.23 defines a full adder using intermediate signals `p` and `g` to compute `s` and `cout`. It produces the same circuit from Figure 4.8, but uses `always/process` statements in place of assignment statements.

These two examples are poor applications of `always/process` statements for modeling combinational logic because they require more lines than the equivalent approach with assignment statements from HDL Examples 4.2 and 4.7. However, `case` and `if` statements are convenient for modeling more complicated combinational logic. `case` and `if` statements must appear within `always/process` statements and are examined in the next sections.

SystemVerilog

In a SystemVerilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, := indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where := is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see HDL Example 4.23). <= can also appear outside `process` statements, where it is also evaluated concurrently.

HDL Example 4.23 FULL ADDER USING `always/process`

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;              // blocking
      g = a & b;              // blocking

      s = p ^ cin;            // blocking
      cout = g | (p & cin);   // blocking
    end
endmodule
```

In this case, `always @(a, b, cin)` would have been equivalent to `always_comb`. However, `always_comb` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

In this case, `process(a, b, cin)` would have been equivalent to `process(all)`. However, `process(all)` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for intermediate variables in combinational logic. This example uses blocking assignments for `p` and `g` so that they get their new values before being used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left hand side of a blocking assignment (:=) in a `process` statement, they must be declared to be `variable` rather than `signal`. The variable declaration appears before the `begin` in the process where the variable is used.

4.5.1 Case Statements

A better application of using the `always/process` statement for combinational logic is a seven-segment display decoder that takes advantage of the `case` statement that must appear inside an `always/process` statement.

As you might have noticed in the seven-segment display decoder of Example 2.10, the design process for large blocks of combinational logic is tedious and prone to error. HDLs offer a great improvement, allowing you to specify the function at a higher level of abstraction, and then automatically synthesize the function into gates. HDL Example 4.24 uses `case` statements to describe a seven-segment display decoder based on its truth table. The `case` statement performs different actions depending on the value of its input. A `case` statement implies combinational logic if all

---

HDL Example 4.24 SEVEN-SEGMENT DISPLAY DECODER

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                 abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                          abcdefg
      when X"0" =>    segments <= "1111110";
      when X"1" =>    segments <= "0110000";
      when X"2" =>    segments <= "1101101";
      when X"3" =>    segments <= "1111001";
      when X"4" =>    segments <= "0110011";
      when X"5" =>    segments <= "1011011";
      when X"6" =>    segments <= "1011111";
      when X"7" =>    segments <= "1110000";
      when X"8" =>    segments <= "1111111";
      when X"9" =>    segments <= "1110011";
      when others =>  segments <= "0000000";
    end case;
  end process;
end;
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the =>, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of `X` for hexadecimal numbers). The `others` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike SystemVerilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like `case` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

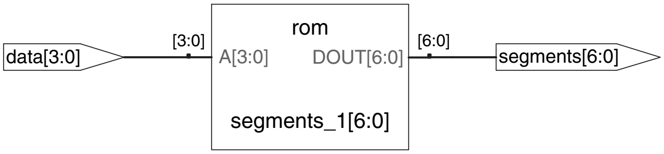


Figure 4.20 `sevenseg` **synthesized circuit**

possible input combinations are defined; otherwise it implies sequential logic, because the output will keep its old value in the undefined cases.

Synplify Premier synthesizes the seven-segment display decoder into a *read-only memory* (*ROM*) containing the 7 outputs for each of the 16 possible inputs. ROMs are discussed further in Section 5.5.6.

If the `default` or `others` clause were left out of the `case` statement, the decoder would have remembered its previous output anytime data were in the range of 10–15. This is strange behavior for hardware.

Ordinary decoders are also commonly written with `case` statements. HDL Example 4.25 describes a 3:8 decoder.

4.5.2 If Statements

`always/process` statements may also contain `if` statements. The `if` statement may be followed by an `else` statement. If all possible input combinations

HDL Example 4.25 3:8 DECODER

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

The `default` statement isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000" =>  y <= "00000001";
      when "001" =>  y <= "00000010";
      when "010" =>  y <= "00000100";
      when "011" =>  y <= "00001000";
      when "100" =>  y <= "00010000";
      when "101" =>  y <= "00100000";
      when "110" =>  y <= "01000000";
      when "111" =>  y <= "10000000";
      when others => y <= "XXXXXXXX";
    end case;
  end process;
end;
```

The `others` clause isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x`, `z`, or `u`.

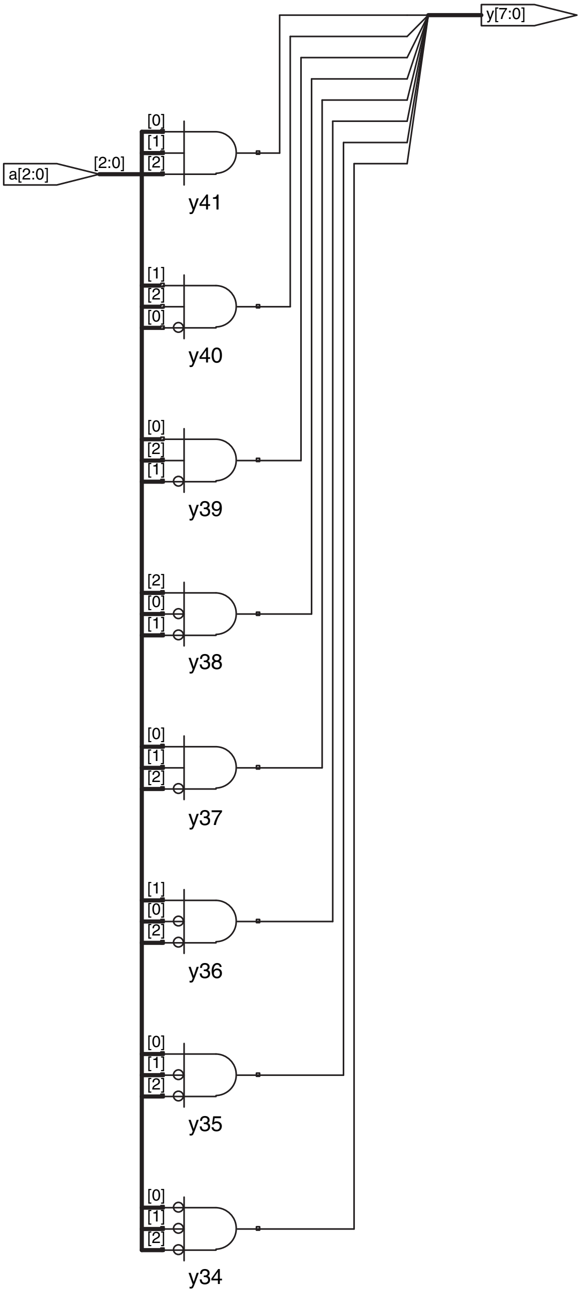

y[7:0]
a[2:0]
[2:0]
[0]
[1]
[2]
y41
[1]
[2]
[0]
y40
[0]
[2]
[1]
y39
[2]
[0]
[1]
y38
[0]
[1]
[2]
y37
[1]
[0]
[2]
y36
[0]
[1]
[2]
y35
[0]
[1]
[2]
y34

Figure 4.21 `decoder3_8` **synthesized circuit**

HDL Example 4.26 PRIORITY CIRCUIT

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside of `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    else            y <= "0000";
    end if;
  end process;
end;
```

Unlike SystemVerilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

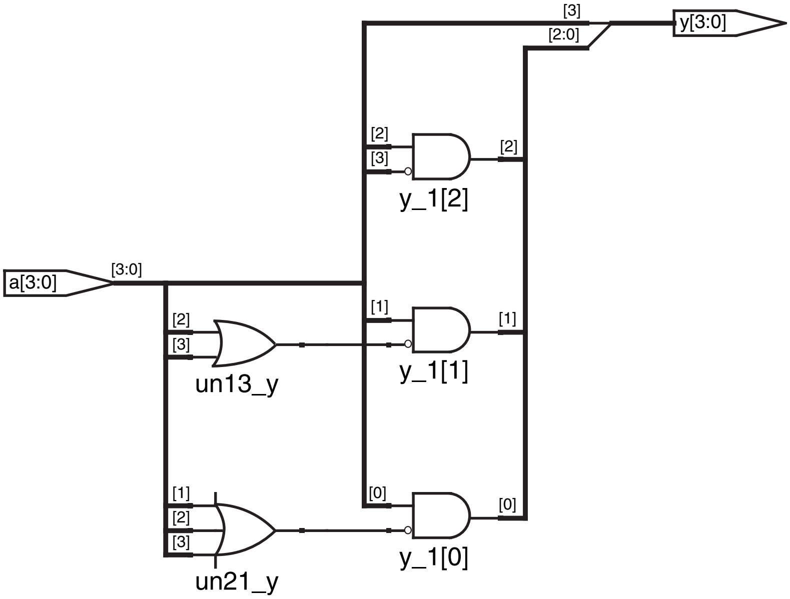


Figure 4.22 `priorityckt` **synthesized circuit**

are handled, the statement implies combinational logic; otherwise, it produces sequential logic (like the latch in Section 4.4.5).

HDL Example 4.26 uses `if` statements to describe a priority circuit, defined in Section 2.4. Recall that an *N*-input priority circuit sets the output TRUE that corresponds to the most significant input that is TRUE.

4.5.3 Truth Tables with Don't Cares

As examined in Section 2.7.3, truth tables may include don't care's to allow more logic simplification. HDL Example 4.27 shows how to describe a priority circuit with don't cares.

Synplify Premier synthesizes a slightly different circuit for this module, shown in Figure 4.23, than it did for the priority circuit in Figure 4.22. However, the circuits are logically equivalent.

HDL Example 4.27 PRIORITY CIRCUIT USING DON'T CARES

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

The `casez` statement acts like a `case` statement except that it also recognizes `?` as don't care.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture dontcare of priority_casez is
begin
  process(all) begin
    case? a is
      when "1---" => y <= "1000";
      when "01--" => y <= "0100";
      when "001-" => y <= "0010";
      when "0001" => y <= "0001";
      when others => y <= "0000";
    end case?;
  end process;
end;
```

The `case?` statement acts like a `case` statement except that it also recognizes `-` as don't care.

4.5.4 Blocking and Nonblocking Assignments

The guidelines on page 206 explain when and how to use each type of assignment. If these guidelines are not followed, it is possible to write code that appears to work in simulation but synthesizes to incorrect hardware. The optional remainder of this section explains the principles behind the guidelines.

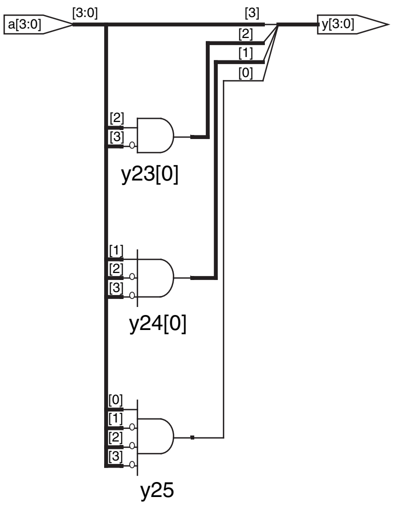


Figure 4.23 `priority_casez` **synthesized circuit**

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

VHDL

1. Use `process(clk)` and nonblocking assignments to model synchronous sequential logic.

```
process(clk) begin
  if rising_edge(clk) then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

```
y <= d0 when s = '0' else d1;
```

3. Use `process(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process(all)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

Combinational Logic*

The full adder from HDL Example 4.23 is correctly modeled using blocking assignments. This section explores how it operates and how it would differ if nonblocking assignments had been used.

Imagine that `a`, `b`, and `cin` are all initially 0. `p`, `g`, `s`, and `cout` are thus 0 as well. At some time, `a` changes to 1, triggering the `always/process` statement. The four blocking assignments evaluate in the order shown here. (In the VHDL code, `s` and `cout` are assigned concurrently.) Note that `p` and `g` get their new values before `s` and `cout` are computed because of the blocking assignments. This is important because we want to compute `s` and `cout` using the new values of `p` and `g`.

1. $\texttt{p} \leftarrow 1 \oplus 0 = 1$
2. $\texttt{g} \leftarrow 1 \bullet 0 = 0$
3. $\texttt{s} \leftarrow 1 \oplus 0 = 1$
4. $\texttt{cout} \leftarrow 0 + 1 \bullet 0 = 0$

In contrast, HDL Example 4.28 illustrates the use of nonblocking assignments.

Now consider the same case of `a` rising from 0 to 1 while `b` and `cin` are 0. The four nonblocking assignments evaluate concurrently:

$$\texttt{p} \leftarrow 1 \oplus 0 = 1 \quad \texttt{g} \leftarrow 1 \bullet 0 = 0 \quad \texttt{s} \leftarrow 0 \oplus 0 = 0 \quad \texttt{cout} \leftarrow 0 + 0 \bullet 0 = 0$$

HDL Example 4.28 FULL ADDER USING NONBLOCKING ASSIGNMENTS

SystemVerilog

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process(all) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Because `p` and `g` appear on the left hand side of a nonblocking assignment in a `process` statement, they must be declared to be `signal` rather than `variable`. The `signal` declaration appears before the `begin` in the `architecture`, not the `process`.

Observe that `s` is computed concurrently with `p` and hence uses the old value of `p`, not the new value. Therefore, `s` remains 0 rather than becoming 1. However, `p` does change from 0 to 1. This change triggers the `always/process` statement to evaluate a second time, as follows:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \bullet 0 = 0 \quad s \leftarrow 1 \oplus 0 = 1 \quad cout \leftarrow 0 + 1 \bullet 0 = 0$$

This time, `p` is already 1, so `s` correctly changes to 1. The nonblocking assignments eventually reach the right answer, but the `always/process` statement had to evaluate twice. This makes simulation slower, though it synthesizes to the same hardware.

Another drawback of nonblocking assignments in modeling combinational logic is that the HDL will produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

Worse yet, some synthesis tools will synthesize the correct hardware even when a faulty sensitivity list causes incorrect simulation. This leads to a mismatch between the simulation results and what the hardware actually does.

SystemVerilog

If the sensitivity list of the `always` statement in HDL Example 4.28 were written as `always @(a, b, cin)` rather than `always_comb`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

VHDL

If the sensitivity list of the `process` statement in HDL Example 4.28 were written as `process(a, b, cin)` rather than `process(all)`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

Sequential Logic*

The synchronizer from HDL Example 4.20 is correctly modeled using nonblocking assignments. On the rising edge of the clock, `d` is copied to `n1` at the same time that `n1` is copied to `q`, so the code properly describes two registers. For example, suppose initially that `d` = 0, `n1` = 1, and `q` = 0. On the rising edge of the clock, the following two assignments occur concurrently, so that after the clock edge, `n1` = 0 and `q` = 1.

$$n1 \leftarrow d = 0 \quad q \leftarrow n1 = 1$$

HDL Example 4.29 tries to describe the same module using blocking assignments. On the rising edge of `clk`, `d` is copied to `n1`. Then this new value of `n1` is copied to `q`, resulting in `d` improperly appearing at both `n1` and `q`. The assignments occur one after the other so that after the clock edge, `q` = `n1` = 0.

1. $n1 \leftarrow d = 0$

2. $q \leftarrow n1 = 0$

HDL Example 4.29 BAD SYNCHRONIZER WITH BLOCKING ASSIGNMENTS

SystemVerilog

```
// Bad implementation of a synchronizer using blocking
// assignments

module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation of a synchronizer using blocking
-- assignment

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

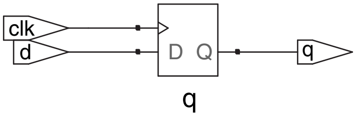


Figure 4.24 `syncbad` **synthesized circuit**

Because `n1` is invisible to the outside world and does not influence the behavior of `q`, the synthesizer optimizes it away entirely, as shown in Figure 4.24.

The moral of this illustration is to exclusively use nonblocking assignment in `always/process` statements when modeling sequential logic. With sufficient cleverness, such as reversing the orders of the assignments, you could make blocking assignments work correctly, but blocking assignments offer no advantages and only introduce the risk of unintended behavior. Certain sequential circuits will not work with blocking assignments no matter what the order.

4.6 FINITE STATE MACHINES

Recall that a finite state machine (FSM) consists of a state register and two blocks of combinational logic to compute the next state and the output given the current state and the input, as was shown in Figure 3.22. HDL descriptions of state machines are correspondingly divided into three parts to model the state register, the next state logic, and the output logic.

HDL Example 4.30 DIVIDE-BY-3 FINITE STATE MACHINE

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate = S1;
      S1:      nextstate = S2;
      S2:      nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `typedef` statement defines `statetype` to be a two-bit `logic` value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100}
statetype;
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality comparison uses /=. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

HDL Example 4.30 describes the divide-by-3 FSM from Section 3.4.2. It provides an asynchronous reset to initialize the FSM. The state register uses the ordinary idiom for flip-flops. The next state and output logic blocks are combinational.

The Synplify Premier synthesis tool just produces a block diagram and state transition diagram for state machines; it does not show the logic

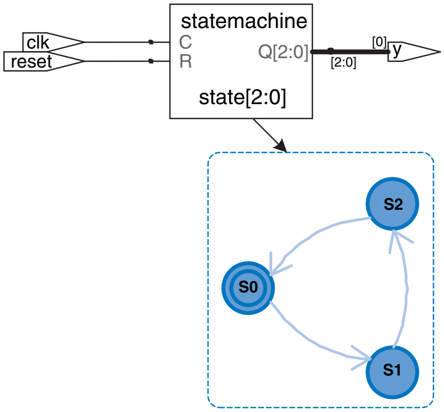


Figure 4.25 `divideby3fsm` **synthesized circuit**

gates or the inputs and outputs on the arcs and states. Therefore, be careful that you have specified the FSM correctly in your HDL code. The state transition diagram in Figure 4.25 for the divide-by-3 FSM is analogous to the diagram in Figure 3.28(b). The double circle indicates that S0 is the reset state. Gate-level implementations of the divide-by-3 FSM were shown in Section 3.4.2.

Notice that the states are named with an enumeration data type rather than by referring to them as binary values. This makes the code more readable and easier to change.

If, for some reason, we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows.

Notice that the synthesis tool uses a 3-bit encoding (`Q[2:0]`) instead of the 2-bit encoding suggested in the SystemVerilog code.

SystemVerilog

```
// output logic
assign y = (state == S0 | state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

The next two examples describe the snail pattern recognizer FSM from Section 3.4.3. The code shows how to use `case` and `if` statements to handle next state and output logic that depend on the inputs as well as the current state. We show both Moore and Mealy modules. In the Moore machine (HDL Example 4.31), the output depends only on the current state, whereas in the Mealy machine (HDL Example 4.32), the output logic depends on both the current state and inputs.

HDL Example 4.31 PATTERN RECOGNIZER MOORE FSM

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

Note how nonblocking assignments (`<=`) are used in the state register to describe sequential logic, whereas blocking assignments (`=`) are used in the next state logic to describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                 state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  --output logic
  y <= '1' when state = S2 else '0';
end;
```

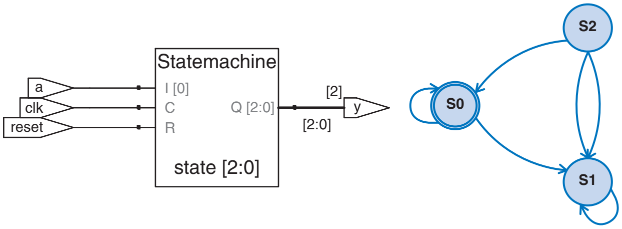


Figure 4.26 `patternMoore` synthesized circuit

HDL Example 4.32 PATTERN RECOGNIZER MEALY FSM

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then               state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

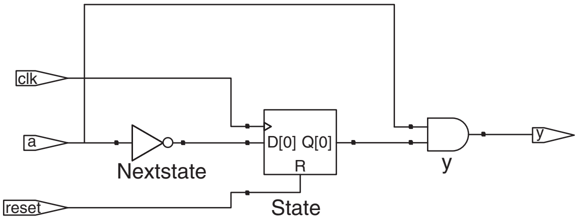


Figure 4.27 `patternMealy` **synthesized circuit**

4.7 DATA TYPES*

This section explains some subtleties about SystemVerilog and VHDL types in more depth.

4.7.1 SystemVerilog

Prior to SystemVerilog, Verilog primarily used two types: `reg` and `wire`. Despite its name, a `reg` signal might or might not be associated with a register. This was a great source of confusion for those learning the language. SystemVerilog introduced the `logic` type to eliminate the confusion; hence, this book emphasizes the `logic` type. This section explains the `reg` and `wire` types in more detail for those who need to read old Verilog code.

In Verilog, if a signal appears on the left hand side of <= or = in an `always` block, it must be declared as `reg`. Otherwise, it should be declared as `wire`. Hence, a `reg` signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an `always` block.

Input and output ports default to the `wire` type unless their type is explicitly defined as `reg`. The following example shows how a flip-flop is described in conventional Verilog. Note that `clk` and `d` default to `wire`, while `q` is explicitly defined as `reg` because it appears on the left hand side of <= in the `always` block.

```
module flop(input              clk,
            input       [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog introduces the `logic` type. `logic` is a synonym for `reg` and avoids misleading users about whether it is actually a flip-flop. Moreover, SystemVerilog relaxes the rules on `assign` statements and hierarchical port instantiations so `logic` can be used outside `always` blocks where a `wire` traditionally would have been required. Thus, nearly all SystemVerilog signals can be `logic`. The exception is that signals with multiple drivers (e.g., a tristate bus) must be declared as a net, as described in HDL Example 4.10. This rule allows SystemVerilog to generate an error message rather than an `x` value when a `logic` signal is accidentally connected to multiple drivers.

The most common type of net is called a `wire` or `tri`. These two types are synonymous, but `wire` is conventionally used when a single driver is present and `tri` is used when multiple drivers are present. Thus, `wire` is obsolete in SystemVerilog because `logic` is preferred for signals with a single driver.

When a `tri` net is driven to a single value by one or more drivers, it takes on that value. When it is undriven, it floats (`z`). When it is driven to a different value (`0`, `1`, or `x`) by multiple drivers, it is in contention (`x`).

There are other net types that resolve differently when undriven or driven by multiple sources. These other types are rarely used, but may be substituted anywhere a `tri` net would normally appear (e.g., for signals with multiple drivers). Each is described in Table 4.7.

Table 4.7 Net Resolution

Net Type	No Driver	Conflicting Drivers
`tri`	`z`	`x`
`trireg`	`previous value`	`x`
`triand`	`z`	`0 if there are any 0`
`trior`	`z`	`1 if there are any 1`
`tri0`	`0`	`x`
`tri1`	`1`	`x`

4.7.2 VHDL

Unlike SystemVerilog, VHDL enforces a strict data typing system that can protect the user from some errors but that is also clumsy at times.

Despite its fundamental importance, the `STD_LOGIC` type is not built into VHDL. Instead, it is part of the `IEEE.STD_LOGIC_1164` library. Thus, every file must contain the library statements shown in the previous examples.

Moreover, `IEEE.STD_LOGIC_1164` lacks basic operations such as addition, comparison, shifts, and conversion to integers for the `STD_ LOGIC_VECTOR` data. These were finally added to the VHDL 2008 standard in the `IEEE.NUMERIC_STD_UNSIGNED` library.

VHDL also has a `BOOLEAN` type with two values: `true` and `false`. `BOOLEAN` values are returned by comparisons (such as the equality comparison, `s = '0'`) and are used in conditional statements such as `when` and `if`. Despite the temptation to believe a `BOOLEAN true` value should be equivalent to a `STD_LOGIC '1'` and `BOOLEAN false` should mean `STD_LOGIC '0'`, these types were not interchangeable prior to VHDL 2008. For example, in old VHDL code, one must write

```
y <= d1 when (s = '1') else d0;
```

while in VHDL 2008, the `when` statement automatically converts `s` from `STD_LOGIC` to `BOOLEAN` so one can simply write

```
y <= d1 when s else d0;
```

Even in VHDL 2008, it is still necessary to write

```
q <= '1' when (state = S2) else '0';
```

instead of

```
q <= (state = S2);
```

because `(state = S2)` returns a `BOOLEAN` result, which cannot be directly assigned to the `STD_LOGIC` signal `y`.

Although we do not declare any signals to be `BOOLEAN`, they are automatically implied by comparisons and used by conditional statements. Similarly, VHDL has an `INTEGER` type that represents both positive and negative integers. Signals of type `INTEGER` span at least the values $-(2^{31} - 1)$ to $2^{31} - 1$. Integer values are used as indices of busses. For example, in the statement

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2, and 3 are integers serving as an index to choose bits of the `a` signal. We cannot directly index a bus with a `STD_LOGIC` or `STD_ LOGIC_ VECTOR` signal. Instead, we must convert the signal to an `INTEGER`. This is demonstrated in the example below for an 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The `TO_INTEGER` function is defined in the `IEEE.NUMERIC_STD_UNSIGNED` library and performs the conversion from `STD_LOGIC_VECTOR` to `INTEGER` for positive (unsigned) values.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(TO_INTEGER(s));
end;
```

VHDL is also strict about `out` ports being exclusively for output. For example, the following code for two- and three-input AND gates is illegal VHDL because `v` is an output and is also used to compute `w`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in STD_LOGIC;
       v, w: out  STD_LOGIC);
end;

architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

VHDL defines a special port type, `buffer`, to solve this problem. A signal connected to a `buffer` port behaves as an output but may also be used within the module. The corrected entity definition follows. Verilog and SystemVerilog do not have this limitation and do not require buffer ports. VHDL 2008 eliminates this restriction by allowing `out` ports to be readable, but this change is not supported by the Synplify CAD tool at the time of this writing.

```
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v: buffer  STD_LOGIC;
       w: out     STD_LOGIC);
end;
```

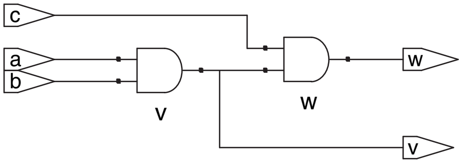


Figure 4.28 `and23` **synthesized circuit**

Most operations such as addition, subtraction, and Boolean logic are identical whether a number is signed or unsigned. However, magnitude comparison, multiplication, and arithmetic right shifts are performed differently for signed two's complement numbers than for unsigned binary numbers. These operations will be examined in Chapter 5. HDL Example 4.33 describes how to indicate that a signal represents a signed number.

HDL Example 4.33 (a) UNSIGNED MULTIPLIER (b) SIGNED MULTIPLIER

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y = a * b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a * b;
endmodule
```

In SystemVerilog, signals are considered unsigned by default. Adding the `signed` modifier (e.g., `logic signed [3:0] a`) causes the signal `a` to be treated as signed.

VHDL

```
-- 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

VHDL uses the `NUMERIC_STD_UNSIGNED` library to perform arithmetic and comparison operations on `STD_LOGIC_VECTOR`s. The vectors are treated as unsigned.

```
use IEEE.NUMERIC_STD_UNSIGNED.all;
```

VHDL also defines `UNSIGNED` and `SIGNED` data types in the `IEEE.NUMERIC_STD` library, but these involve type conversions beyond the scope of this chapter.

4.8 PARAMETERIZED MODULES*

So far all of our modules have had fixed-width inputs and outputs. For example, we had to define separate modules for 4- and 8-bit wide 2:1 multiplexers. HDLs permit variable bit widths using parameterized modules.

HDL Example 4.34 declares a parameterized 2:1 multiplexer with a default width of 8, then uses it to create 8- and 12-bit 4:1 multiplexers.

HDL Example 4.34 PARAMETERIZED *N*-BIT 2:1 MULTIPLEXERS

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

SystemVerilog allows a `#(parameter . . . )` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, in this case called `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#( )` before the instance name, as shown below.

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(. . .)` in defining and overriding parameters.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(1 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

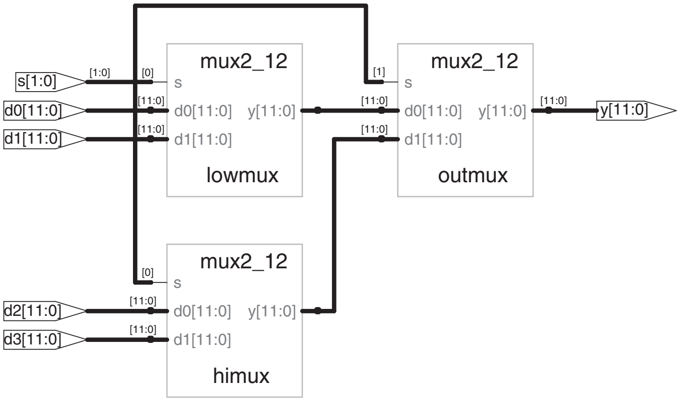

mux2_12
s[1:0]
[1:0]
[0]
s
d0[11:0]
[11:0]
d0[11:0]
y[11:0]
d1[11:0]
[11:0]
d1[11:0]
lowmux
mux2_12
[1]
s
[11:0]
d0[11:0]
y[11:0]
[11:0]
y[11:0]
[11:0]
d1[11:0]
outmux
mux2_12
[0]
s
d2[11:0]
[11:0]
d0[11:0]
y[11:0]
d3[11:0]
[11:0]
d1[11:0]
himux

Figure 4.29 `mux4_12` **synthesized circuit**

HDL Example 4.35 PARAMETERIZED N:2^N DECODER

SystemVerilog

```
module decoder
  #(parameter N = 3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

`2**N` indicates 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in  STD_LOGIC_VECTOR(N-1 downto 0);
        y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

`2**N` indicates 2^N.

HDL Example 4.35 shows a decoder, which is an even better application of parameterized modules. A large N:2^N decoder is cumbersome to specify with `case` statements, but easy using parameterized code that simply sets the appropriate output bit to 1. Specifically, the decoder uses blocking assignments to set all the bits to 0, then changes the appropriate bit to 1.

HDLs also provide `generate` statements to produce a variable amount of hardware depending on the value of a parameter. `generate` supports `for` loops and `if` statements to determine how many of what types of hardware to produce. HDL Example 4.36 demonstrates how to use `generate` statements to produce an N-input AND function from a cascade of two-input AND gates. Of course, a reduction operator would be cleaner and simpler for this application, but the example illustrates the general principle of hardware generators.

Use `generate` statements with caution; it is easy to produce a large amount of hardware unintentionally!

HDL Example 4.36 PARAMETERIZED *N*-INPUT AND GATE

SystemVerilog

```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0] = a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i] = a[i] & x[i-1];
    end

  endgenerate

  assign y = x[width-1];
endmodule
```

The `for` statement loops thrugh `i` = 1, 2, … , `width-1` to produce many consecutive AND gates. The `begin` in a `generate` `for` loop must be followed by a `:` and an arbitrary label (`forloop`, in this case).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
  x(0) <= a(0);
  gen: for i in 1 to width-1 generate
    x(i) <= a(i) and x(i-1);
  end generate;
  y <= x(width-1);
end;
```

The generate loop variable `i` does not need to be declared.

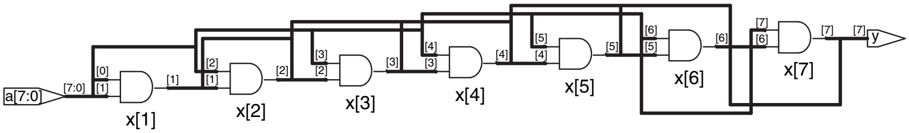


Figure 4.30 `andN` **synthesized circuit**

4.9 TESTBENCHES

Some tools also call the module to be tested the *unit under test* (*UUT*).

A *testbench* is an HDL module that is used to test another module, called the *device under test* (*DUT*). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Consider testing the `sillyfunction` module from Section 4.1.1 that computes $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. This is a simple module, so we can perform exhaustive testing by applying all eight possible test vectors.

HDL Example 4.37 TESTBENCH

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

HDL Example 4.37 demonstrates a simple testbench. It instantiates the DUT, then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order. The user must view the results of the simulation and verify by inspection that the correct outputs are produced. Testbenches are simulated the same as other HDL modules. However, they are not synthesizeable.

Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind; if you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle. A much better approach is to write a self-checking testbench, shown in HDL Example 4.38.

Writing code for each test vector also becomes tedious, especially for modules that require a large number of vectors. An even better approach

HDL Example 4.38 SELF-CHECKING TESTBENCH

SystemVerilog

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0;  #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0;  #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;  #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;  #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

The SystemVerilog `assert` statement checks if a specified condition is true. If not, it executes the `else` statement. The `$error` system task in the `else` statement prints an error message describing the assertion failure. `assert` is ignored during synthesis.

In SystemVerilog, comparison using == or != is effective between signals that do not take on the values of `x` and `z`. Testbenches use the === and !== operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

is to place the test vectors in a separate file. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.

HDL Example 4.39 demonstrates such a testbench. The testbench generates a clock using an `always/process` statement with no sensitivity list, so that it is continuously reevaluated. At the beginning of the simulation, it reads the test vectors from a text file and pulses `reset` for two cycles. Although the clock and reset aren't necessary to test combinational logic, they are included because they would be important to testing a sequential DUT. `example.tv` is a text file containing the inputs and expected output written in binary:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL Example 4.39 TESTBENCH WITH TEST VECTOR FILE

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin  // check result
         $display("Error: inputs = %b", {a, b, c});
         $display(" outputs = %b (%b expected)", y, yexpected);
         errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y:  STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset:  STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
```

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$readmemh` is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs (`a`, `b`, and `c`) and the expected output (`yexpected`) based on the four bits in the current test vector.

The testbench compares the generated output, `y`, with the expected output, `yexpected`, and prints an error if they don't match. `%b` and `%d` indicate to print the values in binary and decimal, respectively. `$display` is a system task to print in the simulator window. For example, `$display("%b %b", y, yexpected);` prints the two values, `y` and `yexpected`, in binary. `%h` prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the `testvectors` array. `$finish` terminates the simulation.

Note that even though the SystemVerilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

```
      read(L, vector_out);
      (a, b, c) <= vector_in(2 downto 0) after 1 ns;
      y_expected <= vector_out after 1 ns;

      -- check results on falling edge
      wait until falling_edge(clk);

      if y /= y_expected then
        report "Error: y = " & std_logic'image(y);
        errors := errors + 1;
      end if;

      vectornum := vectornum + 1;
    end loop;

    -- summarize results at end of simulation
    if (errors = 0) then
      report "NO ERRORS -- " &
            integer'image(vectornum) &
            " tests completed successfully."
            severity failure;
    else
      report integer'image(vectornum) &
            " tests completed, errors = " &
            integer'image(errors)
            severity failure;
    end if;
  end process;
end;
```

The VHDL code uses file reading commands beyond the scope of this chapter, but it gives the sense of what a self-checking testbench looks like in VHDL.

New inputs are applied on the rising edge of the clock, and the output is checked on the falling edge of the clock. Errors are reported as they occur. At the end of the simulation, the testbench prints the total number of test vectors applied and the number of errors detected.

The testbench in HDL Example 4.39 is overkill for such a simple circuit. However, it can easily be modified to test more complex circuits by changing the `example.tv` file, instantiating the new DUT, and changing a few lines of code to set the inputs and check the outputs.

4.10 SUMMARY

Hardware description languages (HDLs) are extremely important tools for modern digital designers. Once you have learned SystemVerilog or VHDL, you will be able to specify digital systems much faster than if you had to draw the complete schematics. The debug cycle is also often much faster, because modifications require code changes instead of tedious schematic rewiring. However, the debug cycle can be much *longer* using HDLs if you don't have a good idea of the hardware your code implies.

HDLs are used for both simulation and synthesis. Logic simulation is a powerful way to test a system on a computer before it is turned into hardware. Simulators let you check the values of signals inside your system that might be impossible to measure on a physical piece of hardware. Logic synthesis converts the HDL code into digital logic circuits.

The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program. The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce. If you don't know what hardware you are implying, you are almost certain not to get what you want. Instead, begin by sketching a block diagram of your system, identifying which portions are combinational logic, which portions are sequential circuits or finite state machines, and so forth. Then write HDL code for each portion, using the correct idioms to imply the kind of hardware you need.

Exercises

The following exercises may be done using your favorite HDL. If you have a simulator available, test your design. Print the waveforms and explain how they prove that it works. If you have a synthesizer available, synthesize your code. Print the generated circuit diagram, and explain why it matches your expectations.

Exercise 4.1 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise1(input  logic a, b, c,
                 output logic y, z);

  assign y = a & b & c | a & b & ~c | a & ~b & c;
  assign z = a & b | ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in   STD_LOGIC;
       y, z:    out  STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or
       (a and not b and c);
  z <= (a and b) or (not a and not b);
end;
```

Exercise 4.2 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise2(input  logic [3:0] a,
                 output logic [1:0] y);
  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(all) begin
    if    a(0) then y <= "11";
    elsif a(1) then y <= "10";
    elsif a(2) then y <= "01";
    elsif a(3) then y <= "00";
    else            y <= a(1 downto 0);
    end if;
  end process;
end;
```

Exercise 4.3 Write an HDL module that computes a four-input XOR function. The input is $a_{3:0}$, and the output is `y`.

Exercise 4.4 Write a self-checking testbench for Exercise 4.3. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works.

Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.5 Write an HDL module called `minority`. It receives three inputs, `a`, `b`, and `c`. It produces one output, `y`, that is TRUE if at least two of the inputs are FALSE.

Exercise 4.6 Write an HDL module for a hexadecimal seven-segment display decoder. The decoder should handle the digits A, B, C, D, E, and F as well as 0–9.

Exercise 4.7 Write a self-checking testbench for Exercise 4.6. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.8 Write an 8:1 multiplexer module called `mux8` with inputs $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, and output `y`.

Exercise 4.9 Write a structural module to compute the logic function, $y = a\bar{b} + \bar{b}\,\bar{c} + \bar{a}bc$, using multiplexer logic. Use the 8:1 multiplexer from Exercise 4.8.

Exercise 4.10 Repeat Exercise 4.9 using a 4:1 multiplexer and as many NOT gates as you need.

Exercise 4.11 Section 4.5.4 pointed out that a synchronizer could be correctly described with blocking assignments if the assignments were given in the proper order. Think of a simple sequential circuit that cannot be correctly described with blocking assignments, regardless of order.

Exercise 4.12 Write an HDL module for an eight-input priority circuit.

Exercise 4.13 Write an HDL module for a 2:4 decoder.

Exercise 4.14 Write an HDL module for a 6:64 decoder using three instances of the 2:4 decoders from Exercise 4.13 and a bunch of three-input AND gates.

Exercise 4.15 Write HDL modules that implement the Boolean equations from Exercise 2.13.

Exercise 4.16 Write an HDL module that implements the circuit from Exercise 2.26.

Exercise 4.17 Write an HDL module that implements the circuit from Exercise 2.27.

Exercise 4.18 Write an HDL module that implements the logic function from Exercise 2.28. Pay careful attention to how you handle don't cares.

Exercise 4.19 Write an HDL module that implements the functions from Exercise 2.35.

Exercise 4.20 Write an HDL module that implements the priority encoder from Exercise 2.36.

Exercise 4.21 Write an HDL module that implements the modified priority encoder from Exercise 2.37.

Exercise 4.22 Write an HDL module that implements the binary-to-thermometer code converter from Exercise 2.38.

Exercise 4.23 Write an HDL module implementing the days-in-month function from Question 2.2.

Exercise 4.24 Sketch the state transition diagram for the FSM described by the following HDL code.

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  logic [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (a ^ b) nextstate = S1;
          else       nextstate = S0;
      S1: if (a & b) nextstate = S2;
          else       nextstate = S0;
      S2: if (a | b) nextstate = S3;
          else       nextstate = S0;
      S3: if (a | b) nextstate = S3;
          else       nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if (a xor b) then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if (a and b) then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

Exercise 4.25 Sketch the state transition diagram for the FSM described by the following HDL code. An FSM of this nature is used in a branch predictor on some microprocessors.

SystemVerilog

```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter  S0  =  5'b00001;
  parameter  SI  =  5'b00010;
  parameter  S2  =  5'b00100;
  parameter  S3  =  5'b01000;
  parameter  S4  =  5'b10000;

  always_ff @(posedge clk, posedge  reset)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken)  nextstate = S1;
          else        nextstate = S0;
      S1: if (taken)  nextstate = S2;
          else        nextstate = S0;
      S2: if (taken)  nextstate = S3;
          else        nextstate = S1;
      S3: if (taken)  nextstate = S4;
          else        nextstate = S2;
      S4: if (taken)  nextstate = S4;
          else        nextstate = S3;
      default:        nextstate = S2;
    endcase

  assign predicttaken = (state  ==  S4)  |
                        (state  ==  S3)  |
                        (state  ==  S2 && back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset:    in   STD_LOGIC;
       taken, back:   in   STD_LOGIC;
       predicttaken:  out  STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

process(all) begin
    case state is
      when S0 => if taken then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken then
                   nextstate => S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if taken then
                   nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken then
                   nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken then
                   nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others =>  nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                  ((state = S4) or (state = S3) or
                  (state = S2 and back = '1'))
  else '0';
end;
```

Exercise 4.26 Write an HDL module for an SR latch.

Exercise 4.27 Write an HDL module for a *JK flip-flop*. The flip-flop has inputs, *clk*, *J*, and *K*, and output *Q*. On the rising edge of the clock, *Q* keeps its old value if $J = K = 0$. It sets *Q* to 1 if $J = 1$, resets *Q* to 0 if $K = 1$, and inverts *Q* if $J = K = 1$.

Exercise 4.28 Write an HDL module for the latch from Figure 3.18. Use one assignment statement for each gate. Specify delays of 1 unit or 1 ns to each gate. Simulate the latch and show that it operates correctly. Then increase the inverter delay. How long does the delay have to be before a race condition causes the latch to malfunction?

Exercise 4.29 Write an HDL module for the traffic light controller from Section 3.4.1.

Exercise 4.30 Write three HDL modules for the factored parade mode traffic light controller from Example 3.8. The modules should be called `controller`, `mode`, and `lights`, and they should have the inputs and outputs shown in Figure 3.33(b).

Exercise 4.31 Write an HDL module describing the circuit in Figure 3.42.

Exercise 4.32 Write an HDL module for the FSM with the state transition diagram given in Figure 3.69 from Exercise 3.22.

Exercise 4.33 Write an HDL module for the FSM with the state transition diagram given in Figure 3.70 from Exercise 3.23.

Exercise 4.34 Write an HDL module for the improved traffic light controller from Exercise 3.24.

Exercise 4.35 Write an HDL module for the daughter snail from Exercise 3.25.

Exercise 4.36 Write an HDL module for the soda machine dispenser from Exercise 3.26.

Exercise 4.37 Write an HDL module for the Gray code counter from Exercise 3.27.

Exercise 4.38 Write an HDL module for the UP/DOWN Gray code counter from Exercise 3.28.

Exercise 4.39 Write an HDL module for the FSM from Exercise 3.29.

Exercise 4.40 Write an HDL module for the FSM from Exercise 3.30.

Exercise 4.41 Write an HDL module for the serial two's complementer from Question 3.2.

Exercise 4.42 Write an HDL module for the circuit in Exercise 3.31.

Exercise 4.43 Write an HDL module for the circuit in Exercise 3.32.

Exercise 4.44 Write an HDL module for the circuit in Exercise 3.33.

Exercise 4.45 Write an HDL module for the circuit in Exercise 3.34. You may use the full adder from Section 4.2.5.

SystemVerilog Exercises

The following exercises are specific to SystemVerilog.

Exercise 4.46 What does it mean for a signal to be declared `tri` in SystemVerilog?

Exercise 4.47 Rewrite the `syncbad` module from HDL Example 4.29. Use nonblocking assignments, but change the code to produce a correct synchronizer with two flip-flops.

Exercise 4.48 Consider the following two SystemVerilog modules. Do they have the same function? Sketch the hardware each one implies.

```
module code1(input  logic clk, a, b, c,
              output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2 (input  logic a, b, c, clk,
               output logic y);
  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

Exercise 4.49 Repeat Exercise 4.48 if the <= is replaced by = in every assignment.

Exercise 4.50 The following SystemVerilog modules show errors that the authors have seen students make in the laboratory. Explain the error in each module and show how to fix it.

(a)
```
module latch(input logic       clk,
             input logic [3:0] d,
             output reg  [3:0] q);

  always @(clk)
    if (clk) q <= d;
endmodule
```

(b)
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

  always @(a)
    begin
      y1 = a & b;
      y2 = a | b;
      y3 = a ^ b;
      y4 = ~(a & b);
      y5 = ~(a | b);
    end
endmodule
```

(c)
```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

    always @(posedge s)
      if (s) y <= d1;
      else   y <= d0;
endmodule
```

(d)
```
module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);

    always @(posedge clk)
      q1 = d1;
      q0 = d0;
endmodule
```

(e)
```
module FSM(input  logic clk,
           input  logic a,
           output logic out1, out2);

    logic state;

    // next state logic and register (sequential)
    always_ff @(posedge clk)
      if (state == 0) begin
          if (a) state  <= 1;
      end else begin
          if (~a) state <= 0;
      end
```

```
  always_comb // output logic (combinational)
    if (state == 0)  out1 = 1;
    else             out2 = 1;
endmodule
```

(f)
```
module priority(input  logic [3:0] a,
                output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

  logic [1:0] state, nextstate;

  parameter  S0  =  2'b00;
  parameter  S1  =  2'b01;
  parameter  S2  =  2'b10;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset)  state <= S0;
    else        state <= nextstate;

  // Next State Logic
  always @(state)
    case (state)
      S0: nextstate  =  S1;
      S1: nextstate  =  S2;
      S2: nextstate  =  S0;
    endcase

  // Output Logic
  assign out = (state == S2);
endmodule
```

(h)
```
module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output tri   [3:0] y);
  tristate t0(d0, s, y);
  tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
                input  logic [3:0] d,
                output logic [3:0] q);
```

```
    always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else       q <= d;

    always @(set)
      if (set)  q <= 1;
endmodule
```

(j)
```
module and3(input  logic a, b, c,
            output logic y);

    logic tmp;

    always @(a, b, c)
    begin
      tmp <= a & b;
      y   <= tmp & c;
    end
endmodule
```

VHDL Exercises

The following exercises are specific to VHDL.

Exercise 4.51 In VHDL, why is it necessary to write

```
q <= '1' when state = S0 else '0';
```

rather than simply

```
q <= (state = S0);
```

Exercise 4.52 Each of the following VHDL modules contains an error. For brevity, only the architecture is shown; assume that the `library` use clause and `entity` declaration are correct. Explain the error and show how to fix it.

(a)
```
architecture synth of latch is
begin
  process(clk) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process(a) begin
    Y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process(clk)
    if rising_edge(clk) then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    end if;
  end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(state) begin
    case state is
      when S0 => nextstate <= S1;
      when S1 => nextstate <= S2;
      when S2 => nextstate <= S0;
    end case;
  end process;

  q <= '1' when state = S0 else '0';
end;
```

(f)
```
architecture struct of mux2 is
   component tristate
     port(a:  in   STD_LOGIC_VECTOR(3 downto 0);
           en: in   STD_LOGIC;
           y:  out STD_LOGIC_VECTOR(3 downto 0));
   end component;

begin
  t0: tristate port map(d0, s, y);
  t1: tristate port map(d1, s, y);
end;
```

(g)
```
architecture asynchronous of floprs is
begin
  process(clk, reset) begin
    if reset then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;

  process(set) begin
    if set then
      q <= '1';
    end if;
  end process;
end;
```

Interview Questions

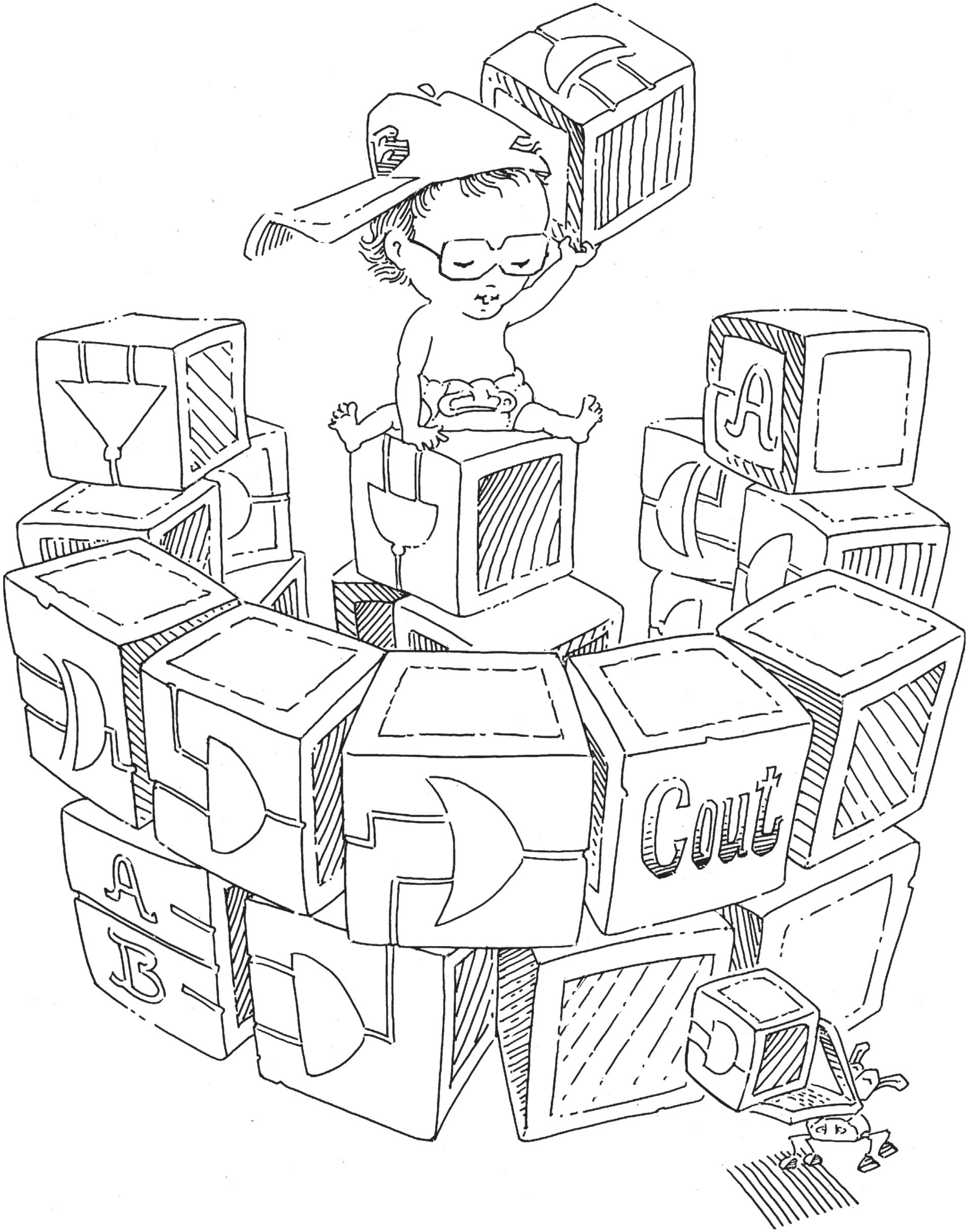
The following exercises present questions that have been asked at interviews for digital design jobs.

Question 4.1 Write a line of HDL code that gates a 32-bit bus called `data` with another signal called `sel` to produce a 32-bit `result`. If `sel` is TRUE, `result` = `data`. Otherwise, `result` should be all 0's.

Question 4.2 Explain the difference between blocking and nonblocking assignments in SystemVerilog. Give examples.

Question 4.3 What does the following SystemVerilog statement do?

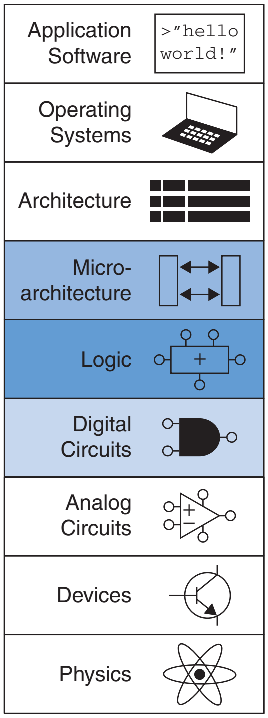
```
result = | (data[15:0] & 16'hC820);
```


A
A
B
Cout

5

Digital Building Blocks

5.1 Introduction
5.2 Arithmetic Circuits
5.3 Number Systems
5.4 Sequential Building Blocks
5.5 Memory Arrays
5.6 Logic Arrays
5.7 Summary
Exercises
Interview Questions



5.1 INTRODUCTION

Up to this point, we have examined the design of combinational and sequential circuits using Boolean equations, schematics, and HDLs. This chapter introduces more elaborate combinational and sequential building blocks used in digital systems. These blocks include arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are not only useful in their own right, but they also demonstrate the principles of hierarchy, modularity, and regularity. The building blocks are hierarchically assembled from simpler components such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes. In Chapter 7, we use many of these building blocks to build a microprocessor.

5.2 ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

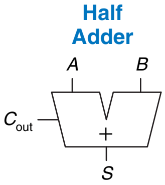
5.2.1 Addition

Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to *N*-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00005-7
© 2013 Elsevier, Inc. All rights reserved.

Half Adder

We begin by building a 1-bit *half adder*. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half adder can be built from an XOR gate and an AND gate.



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

Figure 5.1 **1-bit half adder**

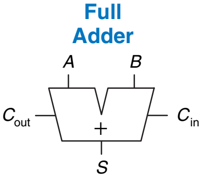
In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output C_{out} of the first column of 1-bit addition and the input C_{in} to the second column of addition. However, the half adder lacks a C_{in} input to accept C_{out} of the previous column. The *full adder*, described in the next section, solves this problem.

```
   1
 0001
+0101
 0110
```

Figure 5.2 **Carry bit**

Full Adder

A *full adder*, introduced in Section 2.1, accepts the carry in C_{in} as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 5.3 **1-bit full adder**

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in C_{in} to produce an N-bit result S and a carry out C_{out}. It is commonly called a *carry propagate adder* (*CPA*) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and prefix adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. It is a good application of modularity and regularity: the full adder module is reused many times to form a larger system. The ripple-carry adder has the disadvantage of being slow when N is large. S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}, as shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder, t_{ripple},

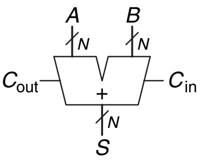


Figure 5.4 **Carry propagate adder**

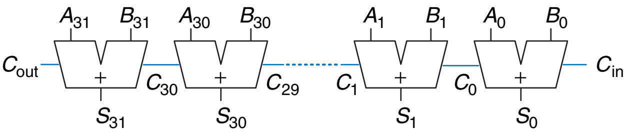


Figure 5.5 **32-bit ripple-carry adder**

grows directly with the number of bits, as given in Equation 5.1, where t_{FA} is the delay of a full adder.

$$t_{\text{ripple}} = N t_{FA} \tag{5.1}$$

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder (*CLA*) is another type of carry propagate adder that solves this problem by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to *look ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder may be divided into eight 4-bit blocks.

CLAs use *generate* (G) and *propagate* (P) signals that describe how a column or block determines the carry out. The ith column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The ith column of an adder is guaranteed to generate a carry C_i if A_i and B_i are both 1. Hence G_i, the generate signal for column i, is calculated as $G_i = A_i B_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The ith column will propagate a carry in, C_{i-1}, if either A_i or B_i is 1. Thus, $P_i = A_i + B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the adder. The ith column of an adder will generate a carry out C_i if it either generates a carry, G_i, or propagates a carry in, $P_i C_{i-1}$. In equation form,

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1} \tag{5.2}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j.

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3\big(G_2 + P_2(G_1 + P_1 G_0)\big) \tag{5.3}$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3 P_2 P_1 P_0 \tag{5.4}$$

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i, using the carry in to the block, C_{j-1}.

$$C_i = G_{i:j} + P_{i:j} C_{j-1} \tag{5.5}$$

Schematics typically show signals flowing from left to right. Arithmetic circuits break this rule because the carries flow from right to left (from the least significant column to the most significant column).

Throughout the ages, people have used many devices to perform arithmetic. Toddlers count on their fingers (and some adults stealthily do too). The Chinese and Babylonians invented the abacus as early as 2400 BC. Slide rules, invented in 1630, were in use until the 1970's, when scientific hand calculators became prevalent. Computers and digital calculators are ubiquitous today. What will be next?

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i, from A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. For a large adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N-bit adder divided into k-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + k t_{FA} \tag{5.6}$$

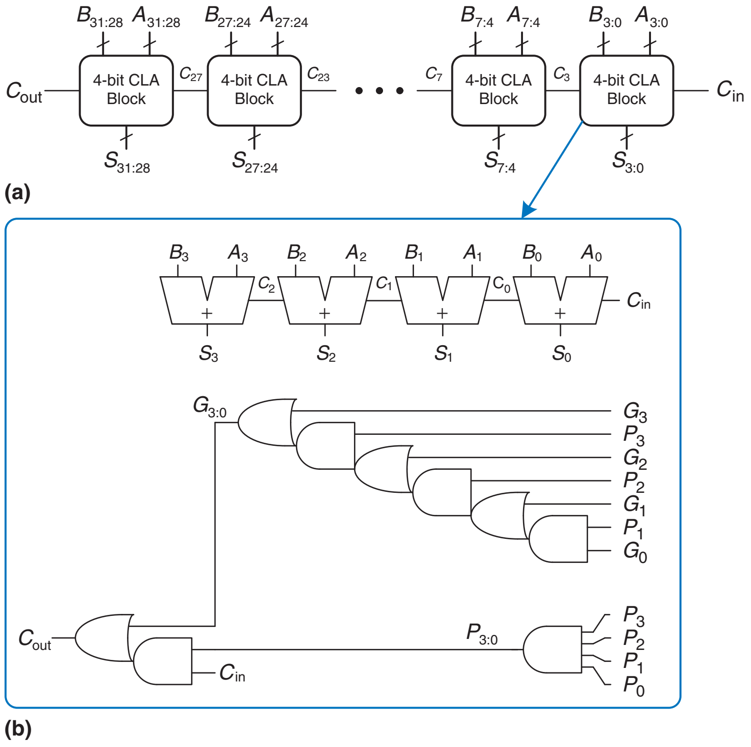

$B_{31:28}$
$A_{31:28}$
$B_{27:24}$
$A_{27:24}$
$B_{7:4}$
$A_{7:4}$
$B_{3:0}$
$A_{3:0}$
C_{out}
4-bit CLA Block
C_{27}
C_{23}
C_7
C_3
C_{in}
$S_{31:28}$
$S_{27:24}$
$S_{7:4}$
$S_{3:0}$
(a)
B_3
A_3
B_2
A_2
B_1
A_1
B_0
A_0
C_2
C_1
C_0
+
S_3
S_2
S_1
S_0
$G_{3:0}$
G_3
P_3
G_2
P_2
G_1
P_1
G_0
$P_{3:0}$
P_3
P_2
P_1
P_0
C_{out}
C_{in}
(b)

Figure 5.6 (a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate P_i and G_i, t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and t_{AND_OR} is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block. For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N.

Example 5.1 RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution: According to Equation 5.1, the propagation delay of the 32-bit ripple-carry adder is 32×300 ps = 9.6 ns.

The CLA has $t_{pg} = 100$ ps, $t_{pg\_block} = 6 \times 100$ ps = 600 ps, and $t_{AND\_OR} = 2 \times 100$ ps = 200 ps. According to Equation 5.6, the propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus 100 ps + 600 ps + $(32/4 - 1) \times 200$ ps + $(4 \times 300$ ps$)$ = 3.3 ns, almost three times faster than the ripple-carry adder.

Prefix Adder*

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5.7}$$

Define column $i = -1$ to hold C_{in}, so $G_{-1} = C_{in}$ and $P_{-1} = 0$. Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i-1$ if the block spanning columns $i-1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation 5.7 as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5.8}$$

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}, G_{0:-1}, G_{1:-1}, G_{2:-1}, \ldots, G_{N-2:-1}$. These signals, along with $P_{-1:-1}, P_{0:-1}, P_{1:-1}, P_{2:-1}, \ldots, P_{N-2:-1}$, are called *prefixes*.

Early computers used ripple-carry adders, because components were expensive and ripple-carry adders used the least hardware. Virtually all modern PCs use prefix adders on critical paths, because transistors are now cheap and speed is of great importance.

Figure 5.7 shows an $N = 16$-bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits *i:k* and from the lower part spanning bits *k*−1:*j*. It combines these parts to form generate and propagate signals for the entire block spanning bits *i:j* using the equations

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{5.9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{5.10}$$

In other words, a block spanning bits *i:j* will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the upper and

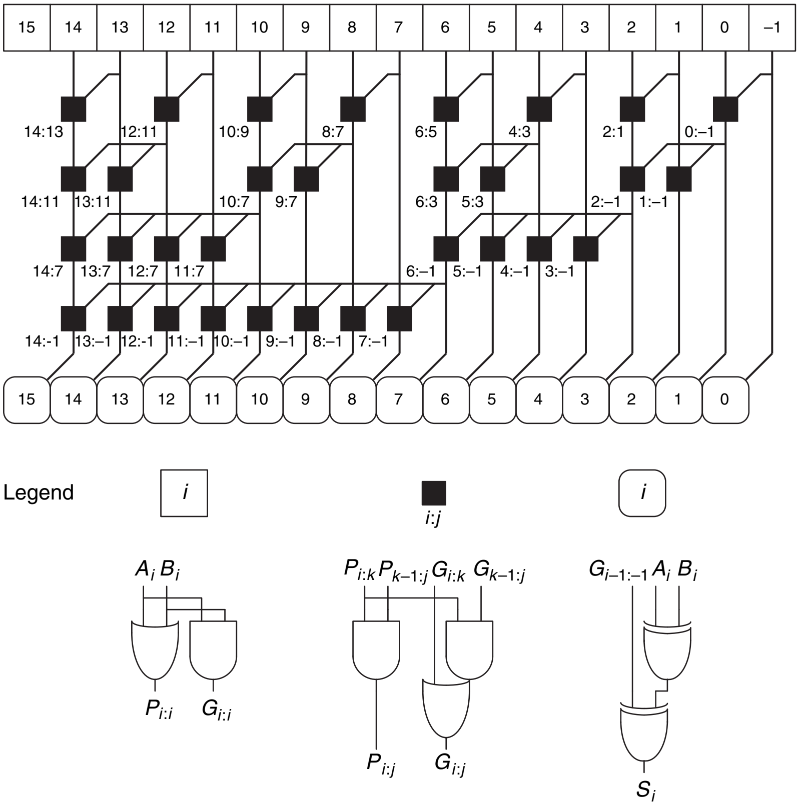

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 –1
14:13 12:11 10:9 8:7 6:5 4:3 2:1 0:–1
14:11 13:11 10:7 9:7 6:3 5:3 2:–1 1:–1
14:7 13:7 12:7 11:7 6:–1 5:–1 4:–1 3:–1
14:-1 13:–1 12:-1 11:–1 10:–1 9:–1 8:–1 7:–1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Legend
i
i:j
i
A_i B_i
P_i:i G_i:i
P_i:k P_k–1:j G_i:k G_k–1:j
P_i:j G_i:j
G_i–1:–1 A_i B_i
S_i

Figure 5.7 **16-bit prefix adder**

lower parts propagate the carry. Finally, the prefix adder computes the sums using Equation 5.8.

In summary, the prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. This speedup is significant, especially for adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-lookahead adder. The network of black cells is called a *prefix tree*.

The general principle of using prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique. With some cleverness, it can be applied to many other types of circuits (see, for example, Exercise 5.7).

The critical path for an N-bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all the prefixes. $G_{i\text{-}1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i. Mathematically, the delay of an N-bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \tag{5.11}$$

where t_{pg_prefix} is the delay of a black prefix cell.

Example 5.2 PREFIX ADDER DELAY

Compute the delay of a 32-bit prefix adder. Assume that each two-input gate delay is 100 ps.

Solution: The propagation delay of each black prefix cell t_{pg_prefix} is 200 ps (i.e., two gate delays). Thus, using Equation 5.11, the propagation delay of the 32-bit prefix adder is 100 ps + $\log_2(32) \times 200$ ps + 100 ps = 1.2 ns, which is about three times faster than the carry-lookahead adder and eight times faster than the ripple-carry adder from Example 5.1. In practice, the benefits are not quite this great, but prefix adders are still substantially faster than the alternatives.

Putting It All Together

This section introduced the half adder, full adder, and three types of carry propagate adders: ripple-carry, carry-lookahead, and prefix adders. Faster adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate adder for a design.

Hardware description languages provide the + operation to specify a CPA. Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the designer's job. HDL Example 5.1 describes a CPA with carries in and out.

HDL Example 5.1 ADDER

SystemVerilog

```
module adder #(parameter N = 8)
             (input  logic [N-1:0] a, b,
              input  logic         cin,
              output logic [N-1:0] s,
              output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(N-1 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result <= ("0" & a) + ("0" & b) + cin;
  s      <= result(N-1 downto 0);
  cout   <= result(N);
end;
```

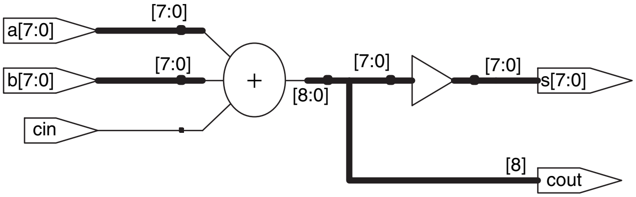


Figure 5.8 Synthesized adder

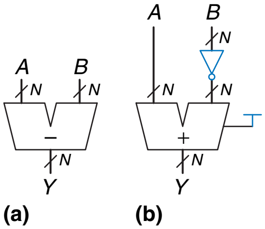


Figure 5.9 Subtractor: (a) symbol, (b) implementation

5.2.2 Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy: flip the sign of the second number, then add. Flipping the sign of a two's complement number is done by inverting the bits and adding 1.

To compute $Y = A - B$, first create the two's complement of B: Invert the bits of B to obtain $\overline{B}$ and add 1 to get $-B = \overline{B} + 1$. Add this quantity to A to get $Y = A + \overline{B} + 1 = A - B$. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{in} = 1$. Figure 5.9 shows the symbol for a subtractor and the underlying hardware for performing $Y = A - B$. HDL Example 5.2 describes a subtractor.

5.2.3 Comparators

A *comparator* determines whether two binary numbers are equal or if one is greater or less than the other. A comparator receives two N-bit binary numbers A and B. There are two common types of comparators.

HDL Example 5.2 SUBTRACTOR

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

  assign y = a – b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity subtractor is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y:    out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

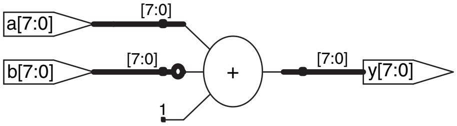


Figure 5.10 **Synthesized subtractor**

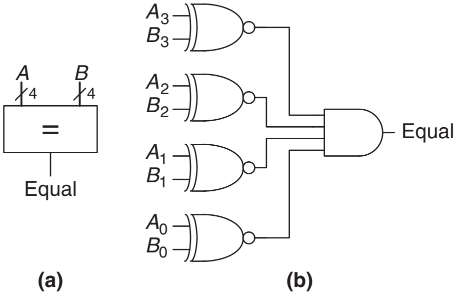


Figure 5.11 **4-bit equality comparator: (a) symbol, (b) implementation**

An *equality comparator* produces a single output indicating whether A is equal to B ($A == B$). A *magnitude comparator* produces one or more outputs indicating the relative values of A and B.

The equality comparator is the simpler piece of hardware. Figure 5.11 shows the symbol and implementation of a 4-bit equality comparator. It first checks to determine whether the corresponding bits in each column of A and B are equal using XNOR gates. The numbers are equal if all of the columns are equal.

Magnitude comparison of signed numbers is usually done by computing $A - B$ and looking at the sign (most significant bit) of the result as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise A is greater than or equal to B. This comparator, however, functions incorrectly upon overflow. Exercises 5.9 and 5.10 explore this limitation and how to fix it.

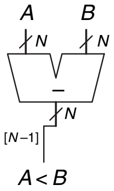


Figure 5.12 ***N*-bit signed comparator**

HDL Example 5.3 COMPARATORS

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity comparators is
  generic(N: integer : = 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparator is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

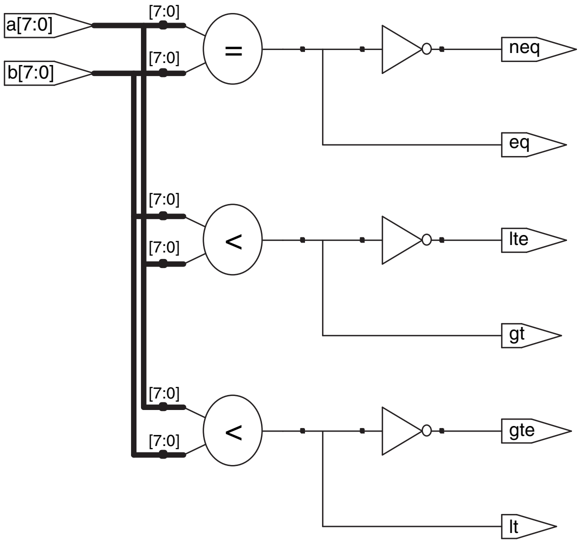


Figure 5.13 **Synthesized comparators**

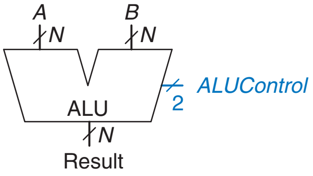


Figure 5.14 **ALU symbol**

HDL Example 5.3 shows how to use various comparison operations for unsigned numbers.

5.2.4 ALU

An *Arithmetic/Logical Unit* (*ALU*) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU

Table 5.1 ALU operations

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

might perform addition, subtraction, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an *N*-bit ALU with *N*-bit inputs and outputs. The ALU receives a 2-bit control signal *ALUControl* that specifies which function to perform. Control signals will generally be shown in blue to distinguish them from the data. Table 5.1 lists typical functions that the ALU can perform.

Figure 5.15 shows an implementation of the ALU. The ALU contains an *N*-bit adder and *N* two-input AND and OR gates. It also contains inverters and a multiplexer to invert input *B* when $ALUControl_0$ is asserted. A 4:1 multiplexer chooses the desired function based on *ALUControl*.

More specifically, if $ALUControl = 00$, the output multiplexer chooses $A + B$. If $ALUControl = 01$, the ALU computes $A - B$. (Recall from Section 5.2.2 that $\overline{B} + 1 = -B$ in two's complement arithmetic. Because $ALUControl_0$ is 1, the adder receives inputs A and $\overline{B}$ and an asserted carry in, causing

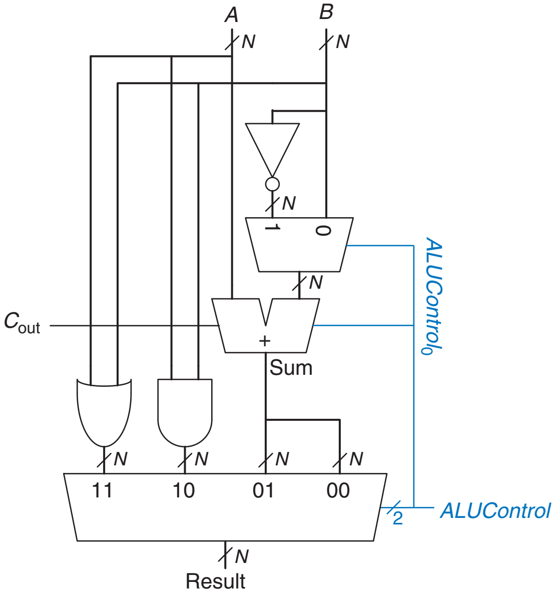


Figure 5.15 ***N*-bit ALU**

it to perform subtraction: $A + \overline{B} + 1 = A - B$.) If $ALUControl = 10$, the ALU computes A AND B. If $ALUControl = 11$, the ALU performs A OR B.

Some ALUs produce extra outputs, called *flags*, that indicate information about the ALU output. Figure 5.16 shows the ALU symbol with a 4-bit *ALUFlags* output. As shown in the schematic of this ALU in Figure 5.17, the *ALUFlags* output is composed of the N, Z, C, and V flags that indicate, respectively, that the ALU output is negative or zero or that the adder produced a carry out or overflowed. Recall that the most significant bit of a two's complement number is 1 if it is negative and 0 otherwise. Thus, the N flag is connected to the most significant bit of the ALU output, $Result_{31}$. The Z flag is asserted when all of the bits of *Result* are 0, as detected by the N-bit NOR gate in Figure 5.17. The C flag is asserted when the adder produces a carry out *and* the ALU is performing addition or subtraction ($ALUControl_1 = 0$).

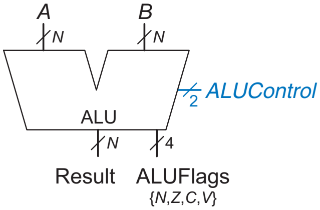

A
B
N
N
ALUControl
2
ALU
N
4
Result
ALUFlags
{N,Z,C,V}

Figure 5.16 ALU symbol with output flags

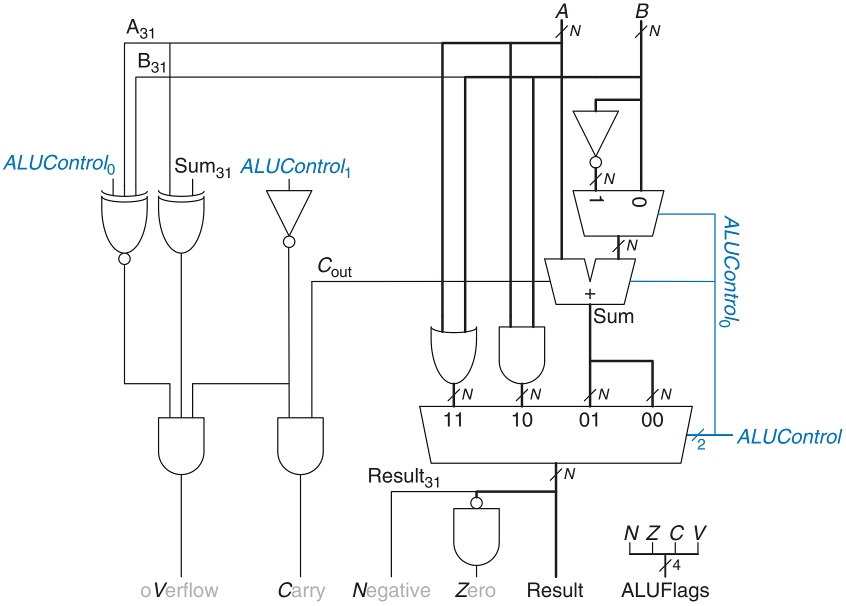

A
B
N
N
A31
B31
ALUControl0
Sum31
ALUControl1
Cout
N
1
0
N
+
Sum
ALUControl0
N
N
N
N
11
10
01
00
2
ALUControl
Result31
N
N Z C V
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags

Figure 5.17 *N*-bit ALU with output flags

Overflow detection, as shown on the left side of Figure 5.17, is trickier. Recall from Section 1.4.6 that overflow occurs when the addition of two same signed numbers produces a result with the opposite sign. So, V is asserted when all three of the following conditions are true: (1) the ALU is performing addition or subtraction ($ALUControl_1 = 0$), (2) A and *Sum* have opposite signs, as detected by the XOR gate, and, as detected by the XNOR gate, (3) either A and B have the same sign and the adder is performing addition ($ALUControl_0 = 0$) or A and B have opposite signs and the adder is performing subtraction ($ALUControl_0 = 1$). The 3-input AND gate detects when all three conditions are true and asserts V.

The HDL for an N-bit ALU with output flags is left to Exercises 5.11 and 5.12. There are many variations on this basic ALU that support other functions, such as XOR or equality comparison.

5.2.5 Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. As the name implies, a shifter shifts a binary number left or right by a specified number of positions. There are several kinds of commonly used shifters:

- **Logical shifter**—shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's.

  Ex: 11001 LSR 2 = 00110; 11001 LSL 2 = 00100

- **Arithmetic shifter**—is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit (msb). This is useful for multiplying and dividing signed numbers (see Sections 5.2.6 and 5.2.7). Arithmetic shift left (ASL) is the same as logical shift left (LSL).

  Ex: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100

- **Rotator**—rotates number in a circle such that empty spots are filled with bits shifted off the other end.

  Ex: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111

An N-bit shifter can be built from N N:1 multiplexers. The input is shifted by 0 to $N - 1$ bits, depending on the value of the $\log_2 N$-bit select lines. Figure 5.18 shows the symbol and hardware of 4-bit shifters. The operators <<, >>, and >>> typically indicate shift left, logical shift right, and arithmetic shift right, respectively. Depending on the value of the 2-bit shift amount $shamt_{1:0}$, the output Y receives the input A shifted by 0 to 3 bits. For all shifters, when $shamt_{1:0} = 00$, $Y = A$. Exercise 5.18 covers rotator designs.

A left shift is a special case of multiplication. A left shift by N bits multiplies the number by 2^N. For example, $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \times 2^4 = 48_{10}$.

An arithmetic right shift is a special case of division. An arithmetic right shift by N bits divides the number by 2^N. For example, 11100_2 $>>> 2 = 11111_2$ is equivalent to $-4_{10}/2^2 = -1_{10}$.

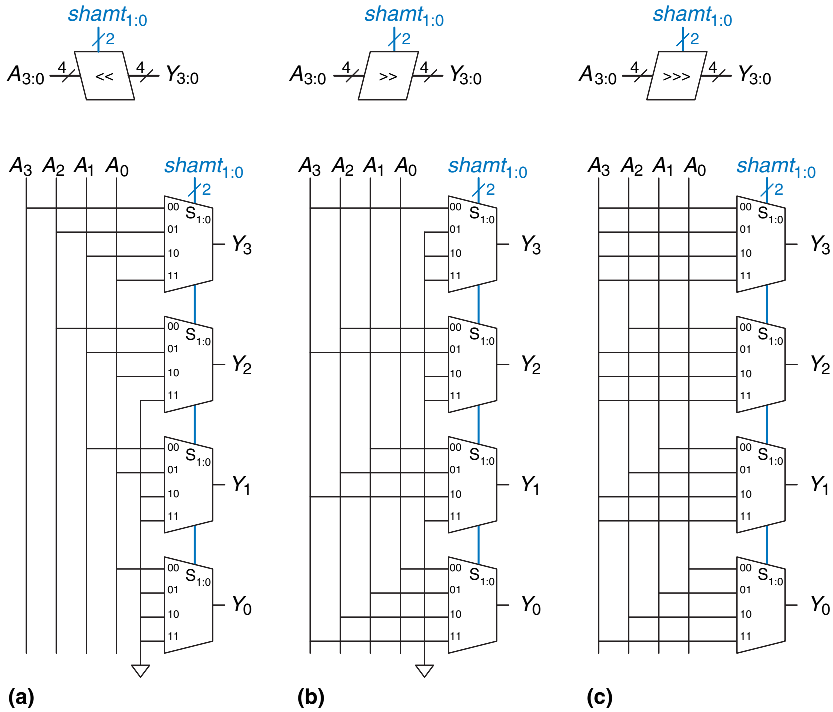

shamt1:0
2
A3:0
4
<<
4
Y3:0
>>
>>>
A3 A2 A1 A0
S1:0
00
01
10
11
Y3
Y2
Y1
Y0
(a)
(b)
(c)

Figure 5.18 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

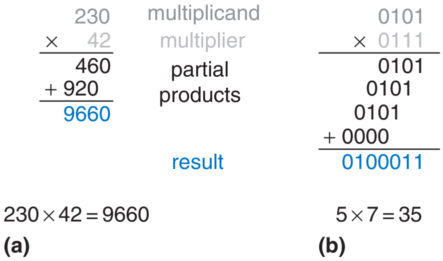

230
× 42
460
+ 920
9660
multiplicand
multiplier
partial
products
result
0101
× 0111
0101
0101
0101
+ 0000
0100011
230 × 42 = 9660
5 × 7 = 35
(a)
(b)

Figure 5.19 Multiplication: (a) decimal, (b) binary

5.2.6 Multiplication*

Multiplication of unsigned binary numbers is similar to decimal multiplication but involves only 1's and 0's. Figure 5.19 compares multiplication in decimal and binary. In both cases, *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result.

In general, an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N$-bit result. The partial products in binary multiplication are either the multiplicand or all 0's. Multiplication of 1-bit binary numbers is equivalent to the AND operation, so AND gates are used to form the partial products.

Signed and unsigned multiplication differ. For example, consider 0xFE × 0xFD. If these 8-bit numbers are interpreted as signed integers, they represent −2 and −3, so the 16-bit product is 0x0006. If these numbers are interpreted as unsigned integers, the 16-bit product is 0xFB06. Notice that in either case, the least significant byte is 0x06.

Figure 5.20 shows the symbol, function, and implementation of an unsigned 4 × 4 multiplier. The unsigned multiplier receives the multiplicand and multiplier, A and B, and produces the product P. Figure 5.20(b) shows how partial products are formed. Each partial product is a single multiplier bit (B_3, B_2, B_1, or B_0) AND the multiplicand bits (A_3, A_2, A_1, A_0). With N-bit operands, there are N partial products and $N-1$ stages of 1-bit adders. For example, for a 4 × 4 multiplier, the partial product of the first row is B_0 AND (A_3, A_2, A_1, A_0). This partial product is added to the shifted second partial product, B_1 AND (A_3, A_2, A_1, A_0). Subsequent rows of AND gates and adders form and add the remaining partial products.

The HDL for signed and unsigned multipliers is in HDL Example 4.33. As with adders, many different multiplier designs with different speed/cost trade-offs exist. Synthesis tools may pick the most appropriate design given the timing constraints.

A *multiply accumulate* operation multiplies two numbers and adds them to a third number, typically the accumulated value. These operations, also called MACs, are often used in *digital signal processing* (DSP) algorithms such as the Fourier transform, which requires a summation of products.

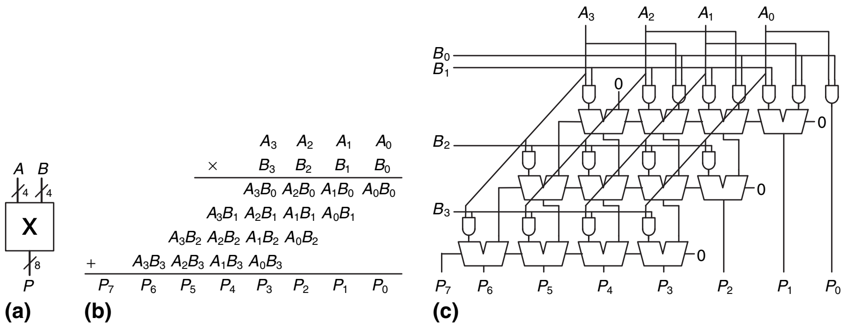


Figure 5.20 4 × 4 multiplier: (a) symbol, (b) function, (c) implementation

5.2.7 Division*

Binary division can be performed using the following algorithm for N-bit unsigned numbers in the range $[0, 2^{N-1}]$:

```
R' = 0
for i = N-1 to 0
   R = {R' << 1, $A_i$}
   D = R - B
   if D < 0 then     $Q_i$ = 0, R' = R    // R < B
   else              $Q_i$ = 1, R' = D    // R ≥ B
R = R'
```

The *partial remainder* R is initialized to 0 ($R' = 0$), and the most significant bit of the dividend A becomes the least significant bit of R ($R = \{R' << 1, A_i\}$). The divisor B is subtracted from this partial remainder to determine whether it fits ($D = R - B$). If the difference D is negative (i.e., the sign bit of D is 1), then the quotient bit Q_i is 0 and the difference is discarded. Otherwise, Q_i is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R, and the process repeats. The result satisfies $\frac{A}{B} = Q + \frac{R}{B}$.

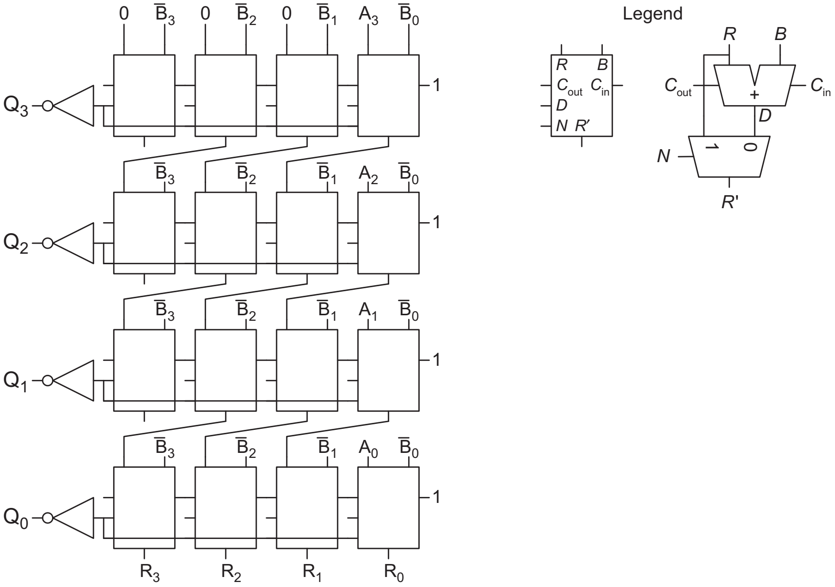


Figure 5.21 Array divider

0
$\overline{B}_3$
0
$\overline{B}_2$
0
$\overline{B}_1$
A_3
$\overline{B}_0$
1
Q_3
A_2
Q_2
A_1
Q_1
A_0
Q_0
R_3
R_2
R_1
R_0
Legend
R
B
C_{out}
C_{in}
D
N
R'
+
1
0

Figure 5.21 shows a schematic of a 4-bit array divider. The divider computes A/B and produces a quotient Q and a remainder R. The legend shows the symbol and schematic for each block in the array divider. Each row performs one iteration of the division algorithm. Specifically, each row calculates the difference $D = R - B$. (Recall that $R + \overline{B} + 1 = R - B$). The signal N indicates whether D is negative. So a row's multiplexer select lines receive the most significant bit of D, which is 1 when the difference is negative. The quotient (Q_i) is 0 when D is negative and 1 otherwise. The multiplexer passes R to the next row if the difference is negative and D otherwise. The following row shifts the new partial remainder left by one bit, appends the next most significant bit of A, and then repeats the process.

The delay of an N-bit array divider increases proportionally to N^2 because the carry must ripple through all N stages in a row before the sign is determined and the multiplexer selects R or D. This repeats for all N rows. Division is a slow and expensive operation in hardware and therefore should be used as infrequently as possible.

5.2.8 Further Reading

Computer arithmetic could be the subject of an entire text. *Digital Arithmetic*, by Ercegovac and Lang, is an excellent overview of the entire field. *CMOS VLSI Design*, by Weste and Harris, covers high-performance circuit designs for arithmetic operations.

5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

5.3.1 Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. For example, Figure 5.22(a) shows a fixed-point number with four integer bits and four fraction bits. Figure 5.22(b) shows the implied binary point in blue, and Figure 5.22(c) shows the equivalent decimal value. The integer bits are called the *high word* and the fraction bits are called the *low word*.

(a) 01101100

(b) 0110.1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Figure 5.22 Fixed-point notation of 6.75 with four integer bits and four fraction bits

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.23 **Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement**

Signed fixed-point numbers can use either two's complement or sign/magnitude notation. Figure 5.23 shows the fixed-point representation of −2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

Like all binary number representations, fixed-point numbers are just a collection of bits. There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number.

Example 5.3 ARITHMETIC WITH FIXED-POINT NUMBERS

Compute $0.75 + -0.625$ using fixed-point numbers.

Solution: First convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because $0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$.

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.24 shows the conversion of −0.625 to fixed-point two's complement notation.

Figure 5.25 shows the fixed-point binary addition and the decimal equivalent for comparison. Note that the leading 1 in the binary fixed-point addition of Figure 5.25(a) is discarded from the 8-bit result.

Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range. Digital signal processing (DSP) applications also often use fixed-point numbers because the computations are faster and consume less power than they would in floating-point.

5.3.2 Floating-Point Number Systems*

Floating-point numbers are analogous to scientific notation. They circumvent the limitation of having a constant number of integer and fraction bits, allowing the representation of very large and very small numbers.

Figure 5.24 **Fixed-point two's complement conversion**

0000.1010	Binary Magnitude
1111.0101	One's Complement
+ 1	Add 1
1111.0110	Two's Complement

Figure 5.25 **Addition: (a) binary fixed-point, (b) decimal equivalent**

(a)	(b)
0000.1100	0.75
+ 1111.0110	+ (–0.625)
10000.0010	0.125

Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.26. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. The decimal point *floats* to the position right after the most significant digit. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

$$\pm M \times B^E$$

Figure 5.26 **Floating-point numbers**

Example 5.4 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution: First convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.27 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.28 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a *biased* exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. For example, for the exponent 7, the biased exponent is $7 + 127 = 134 = 10000110_2$. For the exponent −4, the biased exponent is: $-4 + 127 = 123 = 01111011_2$. Figure 5.29 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of

As may be apparent, there are many reasonable ways to represent floating-point numbers. For many years, computer manufacturers used incompatible floating-point formats. Results from one computer could not directly be interpreted by another computer.

The Institute of Electrical and Electronics Engineers solved this problem by creating the *IEEE 754 floating-point standard* in 1985 defining floating-point numbers. This floating-point format is now almost universally used and is the one discussed in this section.

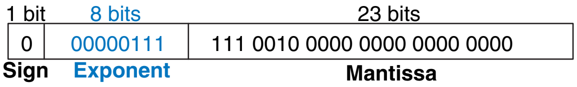


Figure 5.27 **32-bit floating-point version 1**

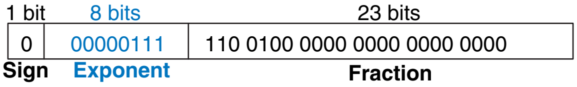


Figure 5.28 **32-bit floating-point version 2**

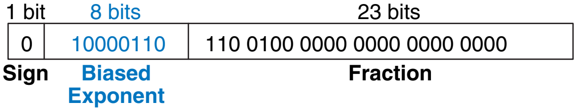


Figure 5.29 **IEEE 754 floating-point notation**

134 (7 + 127). This notation conforms to the IEEE 754 floating-point standard.

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all 1's are reserved for these special cases. Table 5.2 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Single- and Double-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called *single-precision*, *single*, or *float*. The IEEE 754 standard also defines 64-bit *double-precision* numbers (also called *doubles*) that provide greater precision and greater range. Table 5.3 shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$. They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.

Floating-point cannot represent some numbers exactly, like 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999. . . . To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example, the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A–F encodings are not used), and thus decreased performance. So for compute-intensive applications, floating-point is much faster.

Table 5.2 **IEEE 754 floating-point notations for 0, ±∞, and NaN**

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Table 5.3 **Single- and double-precision floating-point formats**

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to $\pm\infty$ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

Figure 5.30 shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}). The result is 8.0625 (1.0000001×2^3). After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the *floating-point unit* (*FPU*), is typically distinct from the *central processing unit* (*CPU*). The infamous *floating-point division* (*FDIV*) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly.

5.4 SEQUENTIAL BUILDING BLOCKS

This section examines sequential building blocks, including counters and shift registers.

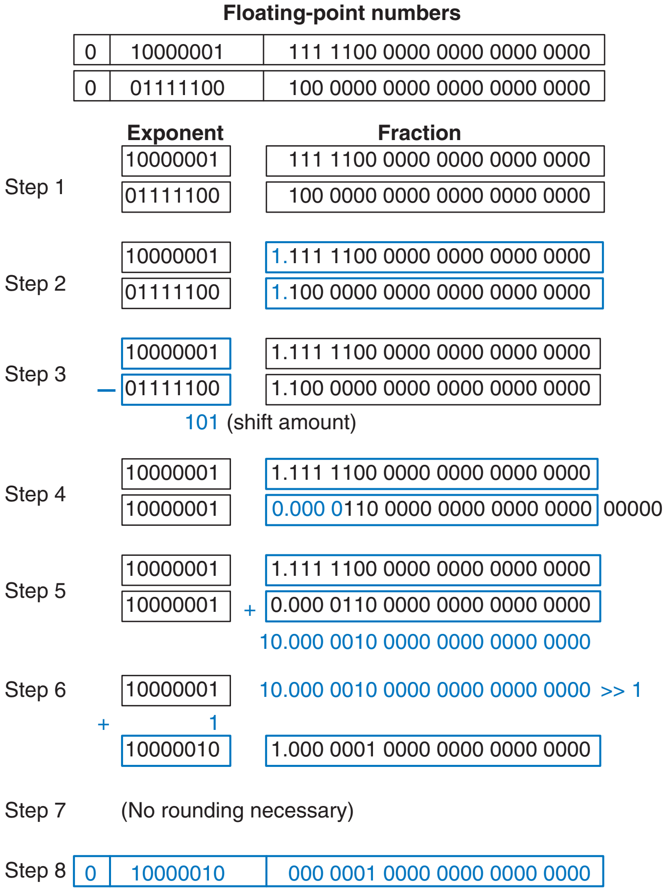

Floating-point numbers
0 | 10000001 | 111 1100 0000 0000 0000 0000
0 | 01111100 | 100 0000 0000 0000 0000 0000
Exponent
Fraction
Step 1
10000001 | 111 1100 0000 0000 0000 0000
01111100 | 100 0000 0000 0000 0000 0000
Step 2
10000001 | 1.111 1100 0000 0000 0000 0000
01111100 | 1.100 0000 0000 0000 0000 0000
Step 3
10000001 | 1.111 1100 0000 0000 0000 0000
– 01111100 | 1.100 0000 0000 0000 0000 0000
101 (shift amount)
Step 4
10000001 | 1.111 1100 0000 0000 0000 0000
10000001 | 0.000 0110 0000 0000 0000 0000 00000
Step 5
10000001 | 1.111 1100 0000 0000 0000 0000
10000001 | + 0.000 0110 0000 0000 0000 0000
10.000 0010 0000 0000 0000 0000
Step 6
10000001 | 10.000 0010 0000 0000 0000 0000 >> 1
+ 1
10000010 | 1.000 0001 0000 0000 0000 0000
Step 7
(No rounding necessary)
Step 8
0 | 10000010 | 000 0001 0000 0000 0000 0000

Figure 5.30 **Floating-point addition**

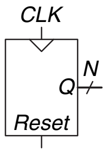

CLK
N
Q
Reset

Figure 5.31 **Counter symbol**

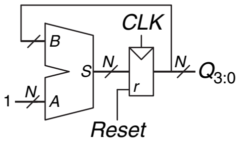

CLK
B
S
N
N
Q3:0
1
N
A
r
Reset

Figure 5.32 ***N*-bit counter**

5.4.1 Counters

An *N*-bit *binary counter*, shown in Figure 5.31, is a sequential arithmetic circuit with clock and reset inputs and an *N*-bit output *Q*. *Reset* initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

Figure 5.32 shows an *N*-bit counter composed of an adder and a resettable register. On each cycle, the counter adds 1 to the value stored in the register. HDL Example 5.4 describes a binary counter with asynchronous reset.

Other types of counters, such as Up/Down counters, are explored in Exercises 5.47 through 5.50.

HDL Example 5.4 COUNTER

SystemVerilog

```
module counter #(parameter N = 8)
               (input  logic clk,
                input  logic reset,
                output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity counter is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       q:          out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process(clk, reset) begin
    if reset then              q <= (OTHERS => '0');
    elsif rising_edge(clk) then q <= q + '1';
    end if;
  end process;
end;
```

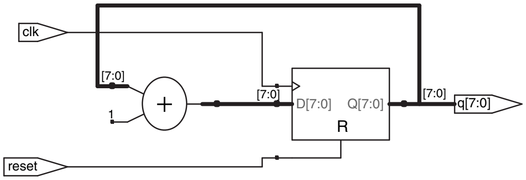


Figure 5.33 Synthesized counter

5.4.2 Shift Registers

A *shift register* has a clock, a serial input S_{in}, a serial output S_{out}, and N parallel outputs $Q_{N-1:0}$, as shown in Figure 5.34. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}. Shift registers can be viewed as *serial-to-parallel converters*. The input is provided serially (one bit at a time) at S_{in}. After N cycles, the past N inputs are available in parallel at Q.

A shift register can be constructed from N flip-flops connected in series, as shown in Figure 5.35. Some shift registers also have a reset signal to initialize all of the flip-flops.

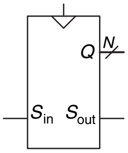


Figure 5.34 Shift register symbol

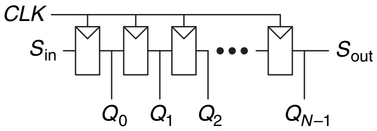


Figure 5.35 Shift register schematic

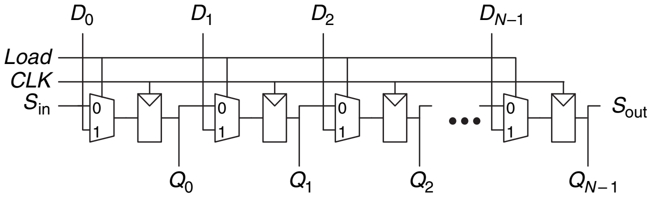


Figure 5.36 Shift register with parallel load

Don't confuse *shift registers* with the *shifters* from Section 5.2.5. Shift registers are sequential logic blocks that shift in a new bit on each clock edge. Shifters are unclocked combinational logic blocks that shift an input by a specified amount.

A related circuit is a *parallel-to-serial* converter that loads *N* bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both serial-to-parallel and parallel-to-serial operations by adding a parallel input $D_{N-1:0}$, and a control signal *Load*, as shown in Figure 5.36. When *Load* is asserted, the flip-flops are loaded in parallel from the *D* inputs. Otherwise, the shift register shifts normally. HDL Example 5.5 describes such a shift register.

Scan Chains*

Shift registers are often used to test sequential circuits using a technique called *scan chains*. Testing combinational circuits is relatively straightforward. Known inputs called *test vectors* are applied, and the outputs are checked against the expected result. Testing sequential circuits is more difficult, because the circuits have state. Starting from a known initial condition, a large number of cycles of test vectors may be needed to put the circuit into a desired state. For example, testing that the most significant bit of a 32-bit counter advances from 0 to 1 requires resetting the counter, then applying 2^{31} (about two billion) clock pulses!

To solve this problem, designers like to be able to directly observe and control all the state of the machine. This is done by adding a test mode in which the contents of all flip-flops can be read out or loaded with desired values. Most systems have too many flip-flops to dedicate individual pins to read and write each flip-flop. Instead, all the flip-flops in the system are connected together into a shift register called a scan chain. In normal operation, the flip-flops load data from their *D* input and ignore the scan chain. In test mode, the flip-flops serially shift their contents out and shift in new contents using S_{in} and S_{out}. The load multiplexer is usually integrated into the flip-flop to produce a *scannable flip-flop*. Figure 5.38 shows the schematic and symbol for a scannable flip-flop and illustrates how the flops are cascaded to build an *N*-bit scannable register.

For example, the 32-bit counter could be tested by shifting in the pattern 011111…111 in test mode, counting for one cycle in normal mode, then shifting out the result, which should be 100000…000. This requires only $32 + 1 + 32 = 65$ cycles.

HDL Example 5.5 SHIFT REGISTER WITH PARALLEL LOAD

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic        clk,
                  input  logic        reset, load,
                  input  logic        sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic        sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(N-1 downto 0);
       q:          out STD_LOGIC_VECTOR(N-1 downto 0);
       sout:       out STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process(clk, reset) begin
    if reset = '1' then   q <= (OTHERS => '0');
    elsif rising_edge(clk) then
      if load then        q <= d;
      else                q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;

  sout <= q(N-1);
end;
```

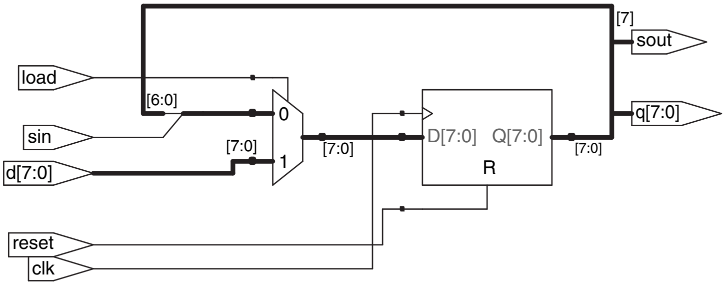

[7]
sout
load
[6:0]
0
q[7:0]
sin
[7:0]
[7:0]
D[7:0]
Q[7:0]
[7:0]
d[7:0]
1
R
reset
clk

Figure 5.37 Synthesized shiftreg

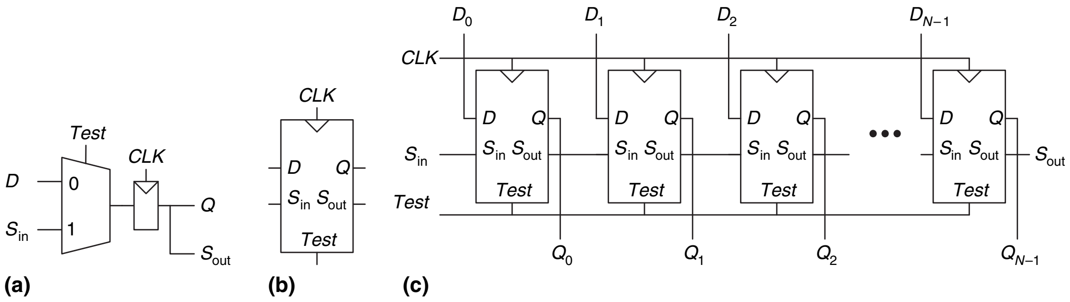

Test
CLK
D
0
Q
S_{in}
1
S_{out}
(a)
CLK
D
Q
S_{in} S_{out}
Test
(b)
D_0
D_1
D_2
D_{N-1}
CLK
D Q
S_{in} S_{out}
Test
S_{in}
Test
S_{out}
Q_0
Q_1
Q_2
Q_{N-1}
(c)

Figure 5.38 Scannable flip-flop: (a) schematic, (b) symbol, and (c) *N*-bit scannable register

5.5 MEMORY ARRAYS

The previous sections introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits. Registers built from flip-flops are a kind of memory that stores small amounts of data. This section describes *memory arrays* that can efficiently store large amounts of data.

The section begins with an overview describing characteristics shared by all memory arrays. It then introduces three types of memory arrays: dynamic random access memory (DRAM), static random access memory (SRAM), and read only memory (ROM). Each memory differs in the way it stores data. The section briefly discusses area and delay trade-offs and shows how memory arrays are used, not only to store data but also to perform logic functions. The section finishes with the HDL for a memory array.

5.5.1 Overview

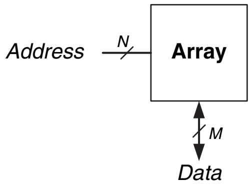


Figure 5.39 **Generic memory array symbol**

Figure 5.39 shows a generic symbol for a memory array. The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is specified by an *Address*. The value read or written is called *Data*. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is called a *word*. Thus, the array contains 2^N M-bit words.

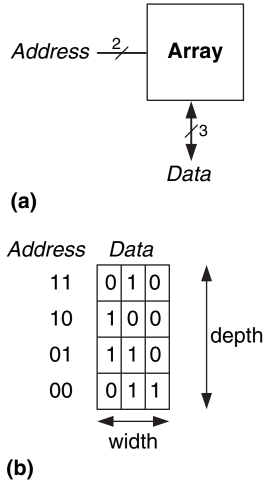


Figure 5.40 **4 × 3 memory array: (a) symbol, (b) function**

Figure 5.40 shows a memory array with two address bits and three data bits. The two address bits specify one of the four rows (data words) in the array. Each data word is three bits wide. Figure 5.40(b) shows some possible contents of the memory array.

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the word size. The size of an array is given as *depth* × *width*. Figure 5.40 is a 4-word × 3-bit array, or simply 4 × 3 array. The symbol for a 1024-word × 32-bit array is shown in Figure 5.41. The total size of this array is 32 kilobits (Kb).

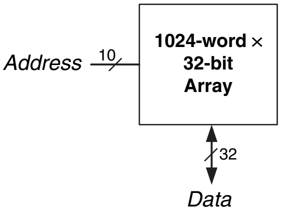


Figure 5.41 **32 Kb array: depth = 2^{10} = 1024 words, width = 32 bits**

Bit Cells

Memory arrays are built as an array of *bit cells*, each of which stores 1 bit of data. Figure 5.42 shows that each bit cell is connected to a *wordline* and a *bitline*. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. The circuitry to store the bit varies with memory type.

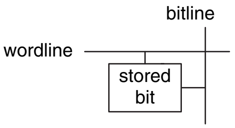


Figure 5.42 **Bit cell**

To read a bit cell, the bitline is initially left floating (Z). Then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then the wordline is turned ON, connecting the bitline to the stored bit. The strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

Organization

Figure 5.43 shows the internal organization of a 4 × 3 memory array. Of course, practical memories are much larger, but the behavior of larger arrays can be extrapolated from the smaller array. In this example, the array stores the data from Figure 5.40(b).

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitlines HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first and then a wordline is asserted, allowing the bitline values to be stored in that row of bit cells. For example, to read *Address* 10, the bitlines are left floating, the decoder asserts $wordline_2$, and the data stored in that row of bit cells (100) reads out onto the *Data* bitlines. To write the value 001 to *Address* 11, the bitlines are driven to the value 001, then $wordline_3$ is asserted and the new value (001) is stored in the bit cells.

Memory Ports

All memories have one or more *ports*. Each port gives read and/or write access to one memory address. The previous examples were all single-ported memories.

Multiported memories can access several addresses simultaneously. Figure 5.44 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address *A1* onto the read data output *RD1*. Port 2 reads the data from address *A2* onto

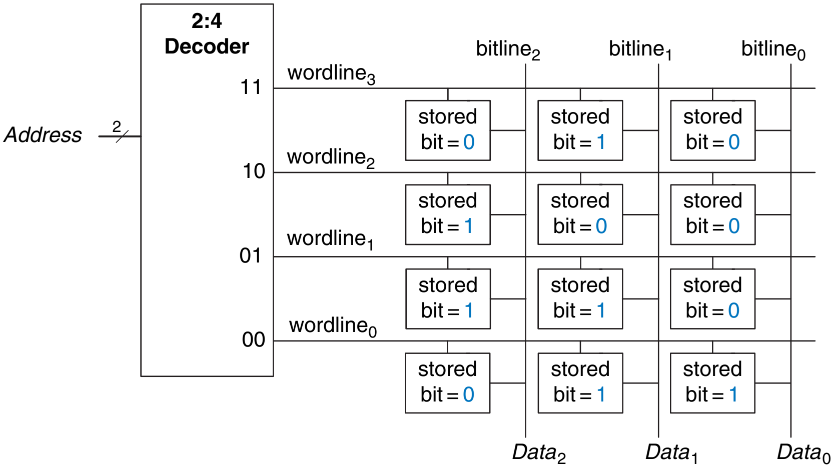

2:4 Decoder
Address
2
11
10
01
00
wordline3
wordline2
wordline1
wordline0
bitline2
bitline1
bitline0
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
Data2
Data1
Data0

Figure 5.43 **4 × 3 memory array**

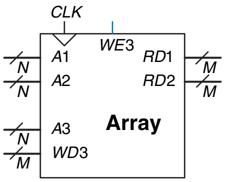

CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Array
N
N
N
M
M
M

Figure 5.44 **Three-ported memory**

RD2. Port 3 writes the data from the write data input *WD3* into address *A3* on the rising edge of the clock if the write enable *WE3* is asserted.

Memory Types

Memory arrays are specified by their size (depth × width) and the number and type of ports. All memory arrays store data as an array of bit cells, but they differ in how they store bits.

Memories are classified based on how they store bits in the bit cell. The broadest classification is *random access memory* (*RAM*) versus *read only memory* (*ROM*). RAM is *volatile*, meaning that it loses its data when the power is turned off. ROM is *nonvolatile*, meaning that it retains its data indefinitely, even without a power source.

RAM and ROM received their names for historical reasons that are no longer very meaningful. RAM is called *random* access memory because any data word is accessed with the same delay as any other. In contrast, a sequential access memory, such as a tape recorder, accesses nearby data more quickly than faraway data (e.g., at the other end of the tape). ROM is called *read only* memory because, historically, it could only be read but not written. These names are confusing, because ROMs are randomly accessed too. Worse yet, most modern ROMs can be written as well as read! The important distinction to remember is that RAMs are volatile and ROMs are nonvolatile.

The two major types of RAMs are *dynamic RAM* (*DRAM*) and *static RAM* (*SRAM*). Dynamic RAM stores data as a charge on a capacitor, whereas static RAM stores data using a pair of cross-coupled inverters. There are many flavors of ROMs that vary by how they are written and erased. These various types of memories are discussed in the subsequent sections.

Robert Dennard, 1932–. Invented DRAM in 1966 at IBM. Although many were skeptical that the idea would work, by the mid-1970s DRAM was in virtually all computers. He claims to have done little creative work until, arriving at IBM, they handed him a patent notebook and said, "put all your ideas in there." Since 1965, he has received 35 patents in semiconductors and microelectronics. (Photo courtesy of IBM.)

5.5.2 Dynamic Random Access Memory (DRAM)

Dynamic RAM (*DRAM*, pronounced "dee-ram") stores a bit as the presence or absence of charge on a capacitor. Figure 5.45 shows a DRAM bit cell. The bit value is stored on a capacitor. The nMOS transistor behaves as a switch that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.

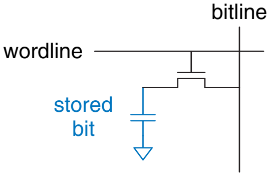


Figure 5.45 DRAM bit cell

As shown in Figure 5.46(a), when the capacitor is charged to V_{DD}, the stored bit is 1; when it is discharged to GND (Figure 5.46(b)), the stored bit is 0. The capacitor node is *dynamic* because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND.

Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to

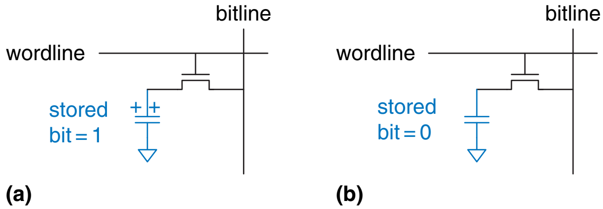


Figure 5.46 **DRAM stored values**

the capacitor. Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

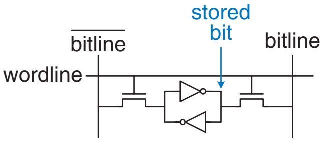


Figure 5.47 **SRAM bit cell**

5.5.3 Static Random Access Memory (SRAM)

Static RAM (*SRAM*, pronounced "es-ram") is *static* because stored bits do not need to be refreshed. Figure 5.47 shows an SRAM bit cell. The data bit is stored on cross-coupled inverters like those described in Section 3.2. Each cell has two outputs, bitline and $\overline{\text{bitline}}$. When the wordline is asserted, both nMOS transistors turn on, and data values are transferred to or from the bitlines. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

5.5.4 Area and Delay

Flip-flops, SRAMs, and DRAMs are all volatile memories, but each has different area and delay characteristics. Table 5.4 shows a comparison of these three types of volatile memory. The data bit stored in a flip-flop is available immediately at its output. But flip-flops take at least 20 transistors to build. Generally, the more transistors a device has, the more area, power, and cost it requires. DRAM latency is longer than that of SRAM because its bitline is not actively driven by a transistor. DRAM must wait for charge to move (relatively) slowly from the capacitor to the bitline. DRAM also fundamentally has lower throughput than SRAM, because it must refresh data

Table 5.4 **Memory comparison**

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

periodically and after a read. DRAM technologies such as *synchronous DRAM* (*SDRAM*) and *double data rate* (*DDR*) SDRAM have been developed to overcome this problem. SDRAM uses a clock to pipeline memory accesses. DDR SDRAM, sometimes called simply DDR, uses both the rising and falling edges of the clock to access data, thus doubling the throughput for a given clock speed. DDR was first standardized in 2000 and ran at 100 to 200 MHz. Later standards, DDR2, DDR3, and DDR4, increased the clock speeds, with speeds in 2015 being over 1 GHz.

Memory latency and throughput also depend on memory size; larger memories tend to be slower than smaller ones if all else is the same. The best memory type for a particular design depends on the speed, cost, and power constraints.

5.5.5 Register Files

Digital systems often use a number of registers to store temporary variables. This group of registers, called a *register file*, is usually built as a small, multiported SRAM array, because it is more compact than an array of flip-flops.

Figure 5.48 shows a 16-register × 32-bit three-ported register file built from a three-ported memory similar to that of Figure 5.44. The register file has two read ports (*A1*/*RD1* and *A2*/*RD2*) and one write port (*A3*/*WD3*). The 4-bit addresses, *A1*, *A2*, and *A3*, can each access all $2^4 = 16$ registers. So, two registers can be read and one register written simultaneously.

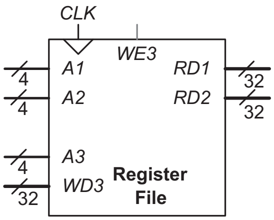


Figure 5.48 **16 × 32 register file with two read ports and one write port**

5.5.6 Read Only Memory

Read only memory (*ROM*) stores a bit as the presence or absence of a transistor. Figure 5.49 shows a simple ROM bit cell. To read the cell, the bitline is weakly pulled HIGH. Then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH. Note that the ROM bit cell is a combinational circuit and has no state to “forget” if power is turned off.

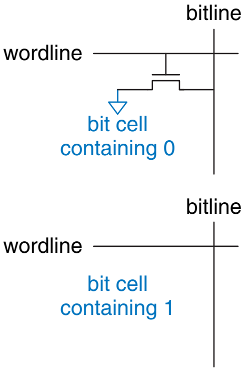


Figure 5.49 **ROM bit cells containing 0 and 1**

The contents of a ROM can be indicated using *dot notation*. Figure 5.50 shows the dot notation for a 4-word × 3-bit ROM containing the data from Figure 5.40. A dot at the intersection of a row (wordline) and a column (bitline) indicates that the data bit is 1. For example, the top wordline has a single dot on $Data_1$, so the data word stored at *Address* 11 is 010.

Conceptually, ROMs can be built using two-level logic with a group of AND gates followed by a group of OR gates. The AND gates produce all possible minterms and hence form a decoder. Figure 5.51 shows the ROM of Figure 5.50 built using a decoder and OR gates. Each dotted row in Figure 5.50 is an input to an OR gate in Figure 5.51.

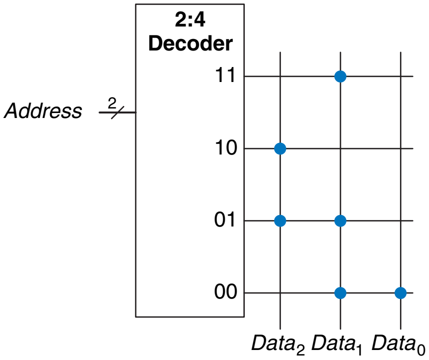

2:4
Decoder
11
10
01
00
Address
2
$Data_2$ $Data_1$ $Data_0$

Figure 5.50 **4 × 3 ROM: dot notation**

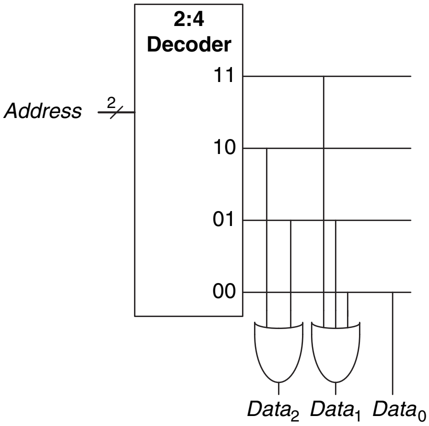

2:4
Decoder
11
10
01
00
Address
2
$Data_2$ $Data_1$ $Data_0$

Figure 5.51 **4 × 3 ROM implementation using gates**

For data bits with a single dot, in this case $Data_0$, no OR gate is needed. This representation of a ROM is interesting because it shows how the ROM can perform any two-level logic function. In practice, ROMs are built from transistors instead of logic gates to reduce their size and cost. Section 5.6.3 explores the transistor-level implementation further.

The contents of the ROM bit cell in Figure 5.49 are specified during manufacturing by the presence or absence of a transistor in each bit cell. A *programmable ROM* (*PROM*, pronounced like the dance) places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.

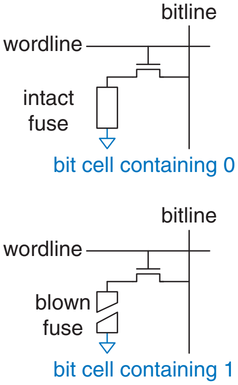

bitline
wordline
intact
fuse
bit cell containing 0
bitline
wordline
blown
fuse
bit cell containing 1

Figure 5.52 **Fuse-programmable ROM bit cell**

Figure 5.52 shows the bit cell for a *fuse-programmable ROM*. The user programs the ROM by applying a high voltage to selectively blow

Fujio Masuoka, 1944–. Received a Ph.D. in electrical engineering from Tohoku University, Japan. Developed memories and high-speed circuits at Toshiba from 1971 to 1994. Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970s. Flash received its name because the process of erasing the memory reminds one of the flash of a camera. Toshiba was slow to commercialize the idea; Intel was first to market in 1988. Flash has grown into a $25 billion per year market. Dr. Masuoka later joined the faculty at Tohoku University and is working to develop a 3-dimensional transistor.

fuses. If the fuse is present, the transistor is connected to GND and the cell holds a 0. If the fuse is destroyed, the transistor is disconnected from ground and the cell holds a 1. This is also called a one-time programmable ROM, because the fuse cannot be repaired once it is blown.

Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND. *Erasable PROMs* (*EPROMs*, pronounced "e-proms") replace the nMOS transistor and fuse with a *floating-gate transistor*. The floating gate is not physically attached to any other wires. When suitable high voltages are applied, electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline (decoder output). When the EPROM is exposed to intense ultraviolet (UV) light for about half an hour, the electrons are knocked off the floating gate, turning the transistor off. These actions are called *programming* and *erasing*, respectively. *Electrically erasable PROMs* (*EEPROMs*, pronounced "e-e-proms" or "double-e proms") and *Flash* memory use similar principles but include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM bit cells are individually erasable; Flash memory erases larger blocks of bits and is cheaper because fewer erasing circuits are needed. In 2015, Flash memory cost about $0.35 per GB, and the price continues to drop by 30 to 40% per year. Flash has become an extremely popular way to store large amounts of data in portable battery-powered systems such as cameras and music players.

In summary, modern ROMs are not really read only; they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write but are nonvolatile.

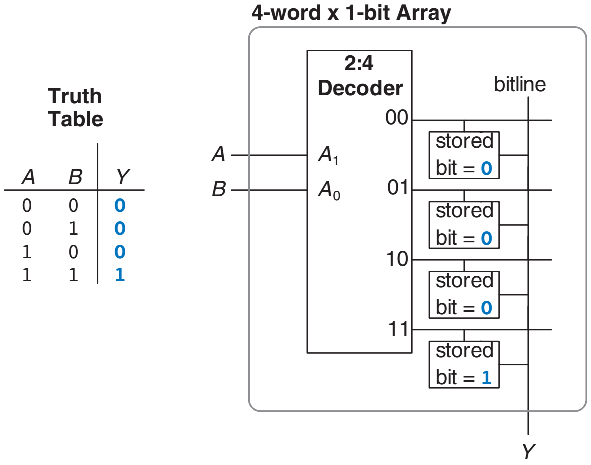
LIBERTY
Lexar

Flash memory drives with Universal Serial Bus (USB) connectors have replaced floppy disks and CDs for sharing files because Flash costs have dropped so dramatically.

5.5.7 Logic Using Memory Arrays

Although they are used primarily for data storage, memory arrays can also perform combinational logic functions. For example, the $Data_2$ output of the ROM in Figure 5.50 is the XOR of the two *Address* inputs. Likewise $Data_0$ is the NAND of the two inputs. A 2^N-word × M-bit memory can perform any combinational function of N inputs and M outputs. For example, the ROM in Figure 5.50 performs three functions of two inputs.

Memory arrays used to perform logic are called *lookup tables* (*LUTs*). Figure 5.53 shows a 4-word × 1-bit memory array used as a lookup table to perform the function $Y = AB$. Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.



Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5.53 **4-word × 1-bit memory array used as a lookup table**

5.5.8 Memory HDL

HDL Example 5.6 describes a 2^N-word × M-bit RAM. The RAM has a synchronous enabled write. In other words, writes occur on the rising edge of the clock if the write enable *we* is asserted. Reads occur immediately. When power is first applied, the contents of the RAM are unpredictable.

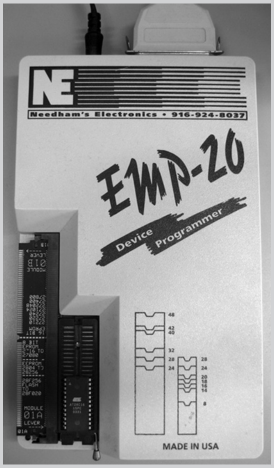
HDL Example 5.7 describes a 4-word × 3-bit ROM. The contents of the ROM are specified in the HDL `case` statement. A ROM as small as this one may be synthesized into logic gates rather than an array. Note that the seven-segment decoder from HDL Example 4.24 synthesizes into a ROM in Figure 4.20.

5.6 LOGIC ARRAYS

Like memory, gates can be organized into regular arrays. If the connections are made programmable, these *logic arrays* can be configured to perform any function without the user having to connect wires in specific ways. The regular structure simplifies design. Logic arrays are mass produced in large quantities, so they are inexpensive. Software tools allow users to map logic designs onto these arrays. Most logic arrays are also reconfigurable, allowing designs to be modified without replacing the hardware. Reconfigurability is valuable during development and is also useful in the field, because a system can be upgraded by simply downloading the new configuration.

This section introduces two types of logic arrays: programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). PLAs, the

Programmable ROMs can be configured with a device programmer like the one shown below. The device programmer is attached to a computer, which specifies the type of ROM and the data values to program. The device programmer blows fuses or injects charge onto a floating gate on the ROM. Thus the programming process is sometimes called *burning* a ROM.



HDL Example 5.6 RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic        clk,
             input  logic        we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
        we:   in  STD_LOGIC;
        adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
        din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
        dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we then mem(TO_INTEGER(adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem(TO_INTEGER(adr));
end;
```

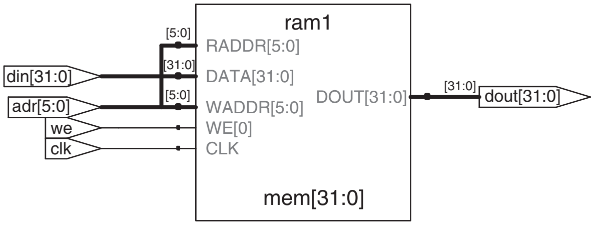


Figure 5.54 **Synthesized ram**

---

older technology, perform only combinational logic functions. FPGAs can perform both combinational and sequential logic.

5.6.1 Programmable Logic Array

Programmable logic arrays (*PLAs*) implement two-level combinational logic in sum-of-products (SOP) form. PLAs are built from an AND array followed by an OR array, as shown in Figure 5.55. The inputs (in true and complementary form) drive an AND array, which produces implicants, which in turn are ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants, and P outputs.

HDL Example 5.7 ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout):

  always_comb
    case(adr)
      2'b00: dout = 3'b011;
      2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(all) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

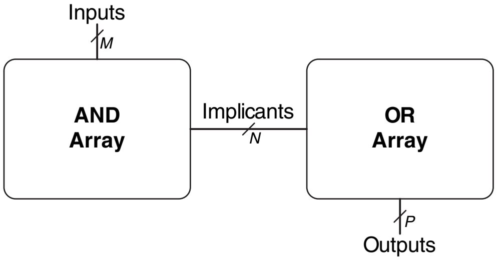


Figure 5.55 ***M*** × ***N*** × ***P*-bit PLA**

Figure 5.56 shows the dot notation for a 3 × 3 × 2-bit PLA performing the functions $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$. Each row in the AND array forms an implicant. Dots in each row of the AND array indicate which literals comprise the implicant. The AND array in Figure 5.56 forms three implicants: $\overline{A}\,\overline{B}C$, $AB\overline{C}$, and $A\overline{B}$. Dots in the OR array indicate which implicants are part of the output function.

Figure 5.57 shows how PLAs can be built using two-level logic. An alternative implementation is given in Section 5.6.3.

ROMs can be viewed as a special case of PLAs. A 2^M-word × N-bit ROM is simply an $M \times 2^M \times N$-bit PLA. The decoder behaves as an AND plane that produces all 2^M minterms. The ROM array behaves as an OR plane that produces the outputs. If the function does not depend on all 2^M minterms, a PLA is likely to be smaller than a ROM. For example, an 8-word × 2-bit ROM is required to perform

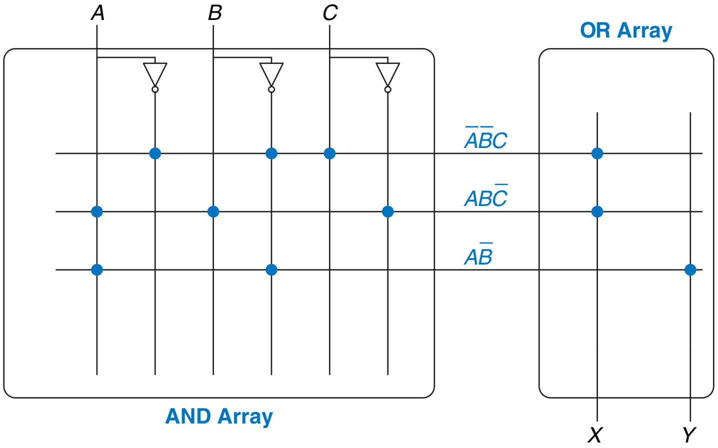

A
B
C
OR Array
$\overline{A}\overline{B}C$
$AB\overline{C}$
$A\overline{B}$
AND Array
X
Y

Figure 5.56 **3 × 3 × 2-bit PLA: dot notation**

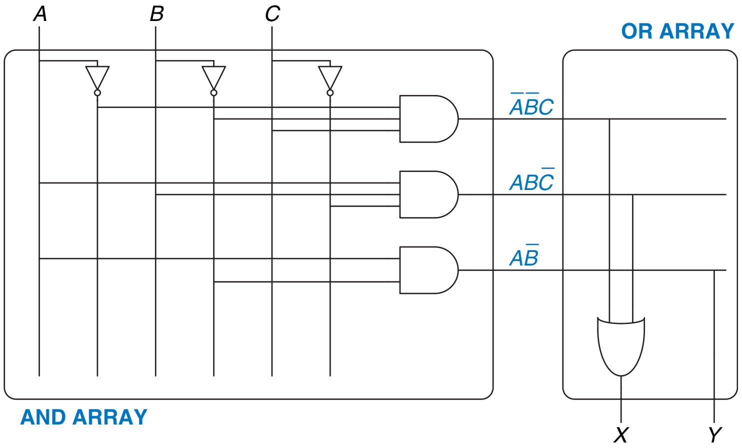

A
B
C
OR ARRAY
$\overline{A}\overline{B}C$
$AB\overline{C}$
$A\overline{B}$
AND ARRAY
X
Y

Figure 5.57 **3 × 3 × 2-bit PLA using two-level logic**

the same functions performed by the 3 × 3 × 2-bit PLA shown in Figures 5.56 and 5.57.

Simple programmable logic devices (*SPLDs*) are souped-up PLAs that add registers and various other features to the basic AND/OR planes. However, SPLDs and PLAs have largely been displaced by FPGAs, which are more flexible and efficient for building large systems.

5.6.2 Field Programmable Gate Array

A *field programmable gate array* (*FPGA*) is an array of reconfigurable gates. Using software programming tools, a user can implement designs on the FPGA using either an HDL or a schematic. FPGAs are more powerful and more flexible than PLAs for several reasons. They can implement both combinational and sequential logic. They can also implement multilevel logic functions, whereas PLAs can only implement two-level logic. Modern FPGAs integrate other useful features such as built-in multipliers, high-speed I/Os, data converters including analog-to-digital converters, large RAM arrays, and processors.

FPGAs are built as an array of configurable *logic elements* (*LEs*), also referred to as *configurable logic blocks* (*CLBs*). Each LE can be configured to perform combinational or sequential functions. Figure 5.58 shows a general block diagram of an FPGA. The LEs are surrounded by *input/output elements* (*IOEs*) for interfacing with the outside world. The IOEs connect LE inputs and outputs to pins on the chip package. LEs can connect to other LEs and IOEs through programmable routing channels.

Two of the leading FPGA manufacturers are Altera Corp. and Xilinx, Inc. Figure 5.59 shows a single LE from Altera's Cyclone IV FPGA introduced in 2009. The key elements of the LE are a 4-input lookup table (LUT) and a 1-bit register. The LE also contains configurable multiplexers to route signals through the LE. The FPGA is configured by specifying the contents of the lookup tables and the select signals for the multiplexers.

FPGAs are the brains of many consumer products, including automobiles, medical equipment, and media devices like MP3 players. The Mercedes Benz S-Class series, for example, has over a dozen Xilinx FPGAs or PLDs for uses ranging from entertainment to navigation to cruise control systems. FPGAs allow for quick time to market and make debugging or adding features late in the design process easier.

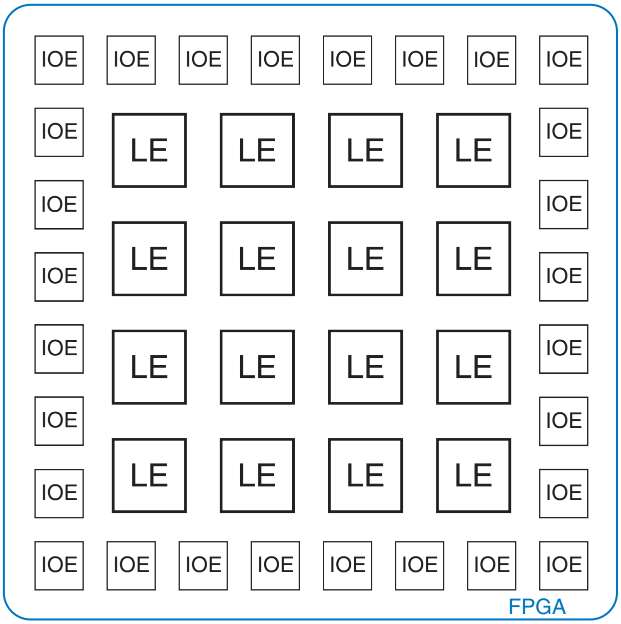


Figure 5.58 General FPGA layout

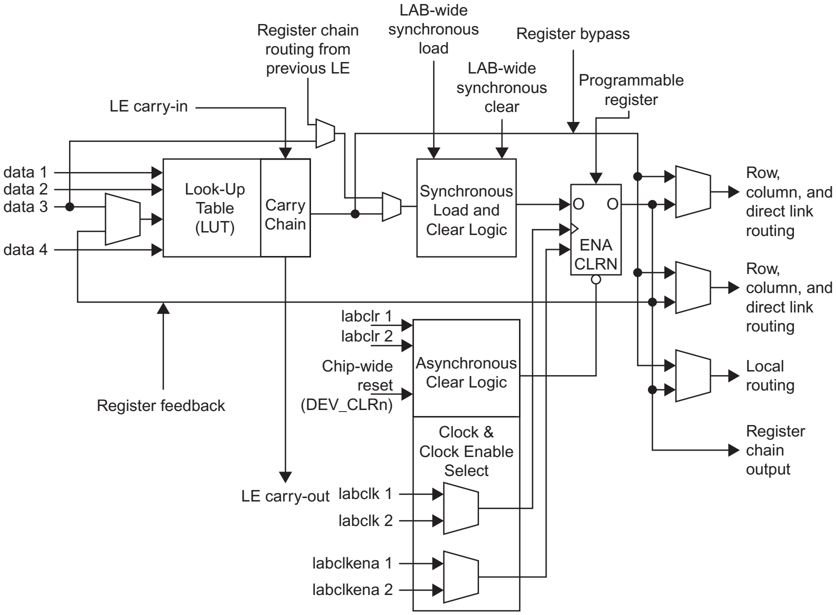

Register chain routing from previous LE
LAB-wide synchronous load
Register bypass
LAB-wide synchronous clear
Programmable register
LE carry-in
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
ENA
CLRN
Row, column, and direct link routing
Row, column, and direct link routing
Local routing
Register chain output
Register feedback
labclr 1
labclr 2
Chip-wide reset (DEV_CLRn)
Asynchronous Clear Logic
Clock & Clock Enable Select
LE carry-out
labclk 1
labclk 2
labclkena 1
labclkena 2

Figure 5.59 **Cyclone IV Logic Element (LE)**
(Reproduced with permission from the Altera Cyclone™ IV Handbook © 2010 Altera Corporation.)

The Cyclone IV LE has one 4-input LUT and one flip-flop. By loading the appropriate values into the lookup table, the LUT can be configured to perform any function of up to four variables. Configuring the FPGA also involves choosing the select signals that determine how the multiplexers route data through the LE and to neighboring LEs and IOEs. For example, depending on the multiplexer configuration, the LUT may receive one of its inputs from either *data 3* or the output of the LE's own register. The other three inputs always come from *data 1*, *data 2*, and *data 4*. The *data 1-4* inputs come from IOEs or the outputs of other LEs, depending on routing external to the LE. The LUT output either goes directly to the LE output for combinational functions, or it can be fed through the flip-flop for registered functions. The flip-flop input comes from its own LUT output, the *data 3* input, or the register output of the previous LE. Additional hardware includes support for addition using the carry chain hardware, other multiplexers for routing, and flip-flop enable and reset. Altera groups 16 LEs together to create a *logic array block* (*LAB*) and provides local connections between LEs within the LAB.

In summary, the Cyclone IV LE can perform one combinational and/or registered function which can involve up to four variables. Other brands of FPGAs are organized somewhat differently, but the same general principles apply. For example, Xilinx's 7-series FPGAs use 6-input LUTs instead of 4-input LUTs.

The designer configures an FPGA by first creating a schematic or HDL description of the design. The design is then synthesized onto the FPGA. The synthesis tool determines how the LUTs, multiplexers, and routing channels should be configured to perform the specified functions. This configuration information is then downloaded to the FPGA. Because Cyclone IV FPGAs store their configuration information in SRAM, they are easily reprogrammed. The FPGA may download its SRAM contents from a computer in the laboratory or from an EEPROM chip when the system is turned on. Some manufacturers include an EEPROM directly on the FPGA or use one-time programmable fuses to configure the FPGA.

Example 5.5 FUNCTIONS BUILT USING LEs

Explain how to configure one or more Cyclone IV LEs to perform the following functions: (a) $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$ (b) $Y = JKLMPQR$; (c) a divide-by-3 counter with binary state encoding (see Figure 3.29(a)). You may show interconnection between LEs as needed.

Solution: (a) Configure two LEs. One LUT computes *X* and the other LUT computes *Y*, as shown in Figure 5.60. For the first LE, inputs *data 1*, *data 2*, and *data 3* are *A*, *B*, and *C*, respectively (these connections are set by the routing channels). *data 4* is a don't care but must be tied to something, so it is tied to 0. For the second LE, inputs *data 1* and *data 2* are *A* and *B*; the other LUT inputs are don't cares and are tied to 0. Configure the final multiplexers to select the combinational outputs from the LUTs to produce *X* and *Y*. In general, a single LE can compute any function of up to four input variables in this fashion.

(*A*) *data 1*	(*B*) *data 2*	(*C*) *data 3*	*data 4*	(*X*) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(*A*) *data 1*	(*B*) *data 2*	*data 3*	*data 4*	(*Y*) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

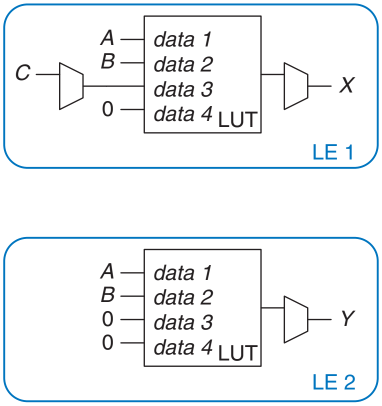


Figure 5.60 LE configuration for two functions of up to four inputs each

(**b**) Configure the LUT of the first LE to compute $X = JKLM$ and the LUT on the second LE to compute $Y = XPQR$. Configure the final multiplexers to select the combinational outputs X and Y from each LE. This configuration is shown in Figure 5.61. Routing channels between LEs, indicated by the dashed blue lines, connect the output of LE 1 to the input of LE 2. In general, a group of LEs can compute functions of N input variables in this manner.

(**c**) The FSM has two bits of state ($S_{1:0}$) and one output (Y). The next state depends on the two bits of current state. Use two LEs to compute the next state from the current state, as shown in Figure 5.62. Use the two flip-flops, one from

(J) data 1	(K) data 2	(L) data 3	(M) data 4	(X) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(P) data 1	(Q) data 2	(R) data 3	(X) data 4	(Y) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

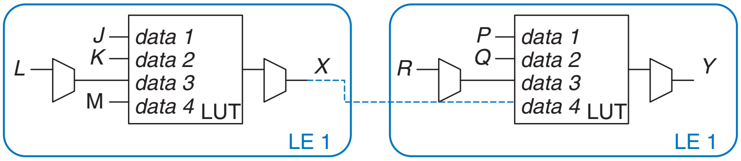

L
J
K
M
data 1
data 2
data 3
data 4
LUT
X
LE 1
R
P
Q
data 1
data 2
data 3
data 4
LUT
Y
LE 1

Figure 5.61 LE configuration for one function of more than four inputs

data 1	data 2	(S_0) data 3	(S_1) data 4	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	data 2	(S_1) data 3	(S_0) data 4	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

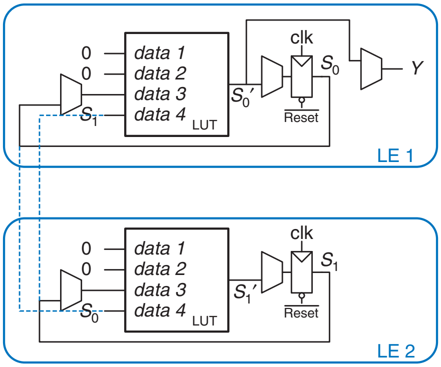

0
0
data 1
data 2
data 3
data 4
LUT
S_1
S_0'
clk
S_0
Reset
Y
LE 1
0
0
data 1
data 2
data 3
data 4
LUT
S_0
S_1'
clk
S_1
Reset
LE 2

Figure 5.62 LE configuration for FSM with two bits of state

each LE, to hold this state. The flip-flops have a reset input that can be connected to an external *Reset* signal. The registered outputs are fed back to the LUT inputs using the multiplexer on *data 3* and routing channels between LEs, as indicated by the dashed blue lines. In general, another LE might be necessary to compute the output Y. However, in this case $Y = S'_0$, so Y can come from LE 1. Hence, the entire FSM fits in two LEs. In general, an FSM requires at least one LE for each bit of state, and it may require more LEs for the output or next state logic if they are too complex to fit in a single LUT.

Example 5.6 LE DELAY

Alyssa P. Hacker is building a finite state machine that must run at 200 MHz. She uses a Cyclone IV FPGA with the following specifications: $t_{LE} = 381$ ps per LE, $t_{\text{setup}} = 76$ ps, and $t_{pcq} = 199$ ps for all flip-flops. The wiring delay between LEs is 246 ps. Assume the hold time for the flip-flops is 0. What is the maximum number of LEs her design can use?

Solution: Alyssa uses Equation 3.13 to solve for the maximum propagation delay of the logic: $t_{pd} \le T_c - (t_{pcq} + t_{\text{setup}})$.

Thus, $t_{pd} = 5 \text{ ns} - (0.199 \text{ ns} + 0.076 \text{ ns})$, so $t_{pd} \le 4.725$ ns. The delay of each LE plus wiring delay between LEs, $t_{LE+wire}$, is $381 \text{ ps} + 246 \text{ ps} = 627$ ps. The maximum number of LEs, N, is $Nt_{LE+wire} \le 4.725$ ns. Thus, $N = 7$.

5.6.3 Array Implementations*

To minimize their size and cost, ROMs and PLAs commonly use pseudo-nMOS or dynamic circuits (see Section 1.7.8) instead of conventional logic gates.

Figure 5.63(a) shows the dot notation for a 4×3-bit ROM that performs the following functions: $X = A \oplus B$, $Y = \overline{A} + B$, and $Z = \overline{A}\,\overline{B}$. These are the same functions as those of Figure 5.50, with the address inputs renamed A and B and the data outputs renamed X, Y, and Z. The pseudo-nMOS implementation is given in Figure 5.63(b). Each decoder output is connected to the gates of the nMOS transistors in its row. Remember that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pull-down (nMOS) network.

Many ROMs and PLAs use dynamic circuits in place of pseudo-nMOS circuits. Dynamic gates turn the pMOS transistor ON for only part of the time, saving power when the pMOS is OFF and the result is not needed. Aside from this, dynamic and pseudo-nMOS memory arrays are similar in design and behavior.

Pull-down transistors are placed at every junction without a dot. The dots from the dot notation diagram of Figure 5.63(a) are left visible in Figure 5.63(b) for easy comparison. The weak pull-up transistors pull the output HIGH for each wordline without a pull-down transistor. For example, when $AB = 11$, the 11 wordline is HIGH and transistors on X and Z turn on and pull those outputs LOW. The Y output has no

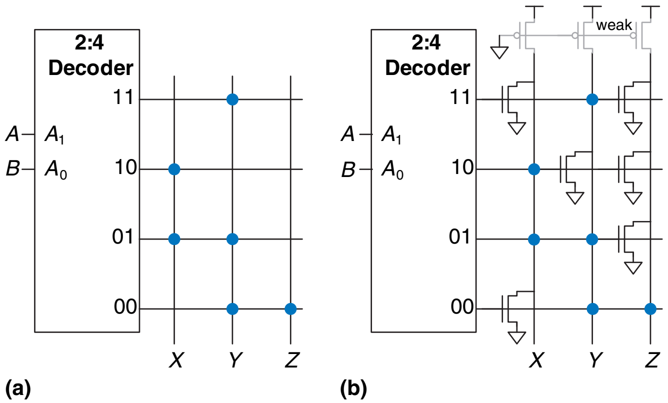

2:4
Decoder
11
10
01
00
A
A1
B
A0
X
Y
Z
(a)
2:4
Decoder
11
10
01
00
A
A1
B
A0
weak
X
Y
Z
(b)

Figure 5.63 ROM implementation: (a) dot notation, (b) pseudo-nMOS circuit

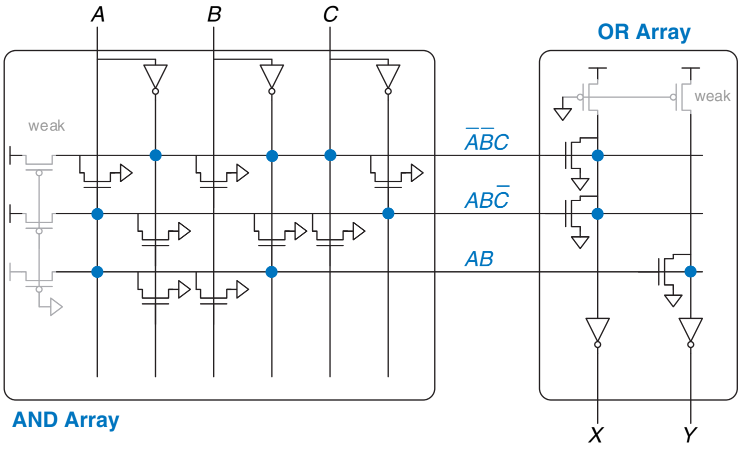

A
B
C
OR Array
weak
weak
$\bar{A}\bar{B}C$
$AB\bar{C}$
AB
AND Array
X
Y

Figure 5.64 $3 \times 3 \times 2$-bit PLA using pseudo-nMOS circuits

transistor connecting to the 11 wordline, so Y is pulled HIGH by the weak pull-up.

PLAs can also be built using pseudo-nMOS circuits, as shown in Figure 5.64 for the PLA from Figure 5.56. Pull-down (nMOS) transistors are placed on the *complement* of dotted literals in the AND array and on dotted rows in the OR array. The columns in the OR array are sent through an inverter before they are fed to the output bits. Again, the blue dots from the dot notation diagram of Figure 5.56 are left visible in Figure 5.64 for easy comparison.

5.7 SUMMARY

This chapter introduced digital building blocks used in many digital systems. These blocks include arithmetic circuits such as adders, subtractors, comparators, shifters, multipliers, and dividers; sequential circuits such as counters and shift registers; and arrays for memory and logic. The chapter also explored fixed-point and floating-point representations of fractional numbers. In Chapter 7, we use these building blocks to build a microprocessor.

Adders form the basis of most arithmetic circuits. A half adder adds two 1-bit inputs, A and B, and produces a sum and a carry out. A full adder extends the half adder to also accept a carry in. N full adders can be cascaded to form a carry propagate adder (CPA) that adds two N-bit numbers. This type of CPA is called a ripple-carry adder because the carry ripples through each of the full adders. Faster CPAs can be constructed using lookahead or prefix techniques.

A subtractor negates the second input and adds it to the first. A magnitude comparator subtracts one number from another and determines the relative value based on the sign of the result. A multiplier forms partial products using AND gates, then sums these bits using full adders. A divider repeatedly subtracts the divisor from the partial remainder and checks the sign of the difference to determine the quotient bits. A counter uses an adder and a register to increment a running count.

Fractional numbers are represented using fixed-point or floating-point forms. Fixed-point numbers are analogous to decimals, and floating-point numbers are analogous to scientific notation. Fixed-point numbers use ordinary arithmetic circuits, whereas floating-point numbers require more elaborate hardware to extract and process the sign, exponent, and mantissa.

Large memories are organized into arrays of words. The memories have one or more ports to read and/or write the words. Volatile memories, such as SRAM and DRAM, lose their state when the power is turned off. SRAM is faster than DRAM but requires more transistors. A register file is a small multiported SRAM array. Nonvolatile memories, called ROMs, retain their state indefinitely. Despite their names, most modern ROMs can be written.

Arrays are also a regular way to build logic. Memory arrays can be used as lookup tables to perform combinational functions. PLAs are composed of dedicated connections between configurable AND and OR arrays; they only implement combinational logic. FPGAs are composed of many small lookup tables and registers; they implement combinational and sequential logic. The lookup table contents and their interconnections can be configured to perform any logic function. Modern FPGAs are easy to reprogram and are large and cheap enough to build highly sophisticated digital systems, so they are widely used in low- and medium-volume commercial products as well as in education.

Exercises

Exercise 5.1 What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

(b) a carry-lookahead adder with 4-bit blocks

(c) a prefix adder

Exercise 5.2 Design two adders: a 64-bit ripple-carry adder and a 64-bit carry-lookahead adder with 4-bit blocks. Use only two-input gates. Each two-input gate is 15 μm^2, has a 50 ps delay, and has 20 fF of total gate capacitance. You may assume that the static power is negligible.

(a) Compare the area, delay, and power of the adders (operating at 100 MHz and 1.2 V).

(b) Discuss the trade-offs between power, area, and delay.

Exercise 5.3 Explain why a designer might choose to use a ripple-carry adder instead of a carry-lookahead adder.

Exercise 5.4 Design the 16-bit prefix adder of Figure 5.7 in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.5 The prefix network shown in Figure 5.7 uses black cells to compute all of the prefixes. Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k-1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

Exercise 5.6 The prefix network shown in Figure 5.7 is not the only way to calculate all of the prefixes in logarithmic time. The *Kogge-Stone* network is another common prefix network that performs the same function using a different connection of black cells. Research Kogge-Stone adders and draw a schematic similar to Figure 5.7 showing the connection of black cells in a Kogge-Stone adder.

Exercise 5.7 Recall that an N-input priority encoder has $\log_2 N$ outputs that encodes which of the N inputs gets priority (see Exercise 2.36).

(a) Design an N-input priority encoder that has delay that increases logarithmically with N. Sketch your design and give the delay of the circuit in terms of the delay of its circuit elements.

(b) Code your design in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.8 Design the following comparators for 32-bit unsigned numbers. Sketch the schematics.

(a) not equal

(b) greater than or equal to

(c) less than

Exercise 5.9 Consider the signed comparator of Figure 5.12.

(a) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator correctly computes $A < B$.

(b) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator incorrectly computes $A < B$.

(c) In general, when does the N-bit signed comparator operate incorrectly?

Exercise 5.10 Modify the N-bit signed comparator of Figure 5.12 to correctly compute $A < B$ for all N-bit signed inputs A and B.

Exercise 5.11 Design the 32-bit ALU shown in Figure 5.15 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.12 Design the 32-bit ALU shown in Figure 5.17 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.13 Write a testbench to test the 32-bit ALU from Exercise 5.11. Then use it to test the ALU. Include any test vector files necessary. Be sure to test enough corner cases to convince a reasonable skeptic that the ALU functions correctly.

Exercise 5.14 Repeat Exercise 5.13 for the ALU from Exercise 5.12.

Exercise 5.15 Build an Unsigned Comparison Unit that compares two unsigned numbers A and B. The unit's input is the *ALUFlags* signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are HS, LS, HI, and LO, which indicate that A is higher than or the same as (HS), lower than or the same as (LS), higher (HI), or lower (LO) than B.

(a) Write minimal equations for HS, LS, HI, and LO in terms of N, Z, C, and V.

(b) Sketch circuits for HS, LS, HI, and LO.

Exercise 5.16 Build a Signed Comparison Unit that compares two signed numbers A and B. The unit's input is the *ALUFlags* signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are GE, LE, GT, and LT, which indicate that A is greater than or equal to (GE), less than or equal to (LE), greater than (GT), or less than (LT) B.

(a) Write minimal equations for GE, LE, GT, and LT in terms of N, Z, C, and V.

(b) Sketch circuits for GE, LE, GT, and LT.

Exercise 5.17 Design a shifter that always shifts a 32-bit input left by 2 bits. The input and output are both 32 bits. Explain the design in words and sketch a schematic. Implement your design in your favorite HDL.

Exercise 5.18 Design 4-bit left and right rotators. Sketch a schematic of your design. Implement your design in your favorite HDL.

Exercise 5.19 Design an 8-bit left shifter using only 24 2:1 multiplexers. The shifter accepts an 8-bit input A and a 3-bit shift amount, $shamt_{2:0}$. It produces an 8-bit output Y. Sketch the schematic.

Exercise 5.20 Explain how to build any N-bit shifter or rotator using only $N\log_2 N$ 2:1 multiplexers.

Exercise 5.21 The *funnel shifter* in Figure 5.65 can perform any N-bit shift or rotate operation. It shifts a $2N$-bit input right by k bits. The output Y is the N least significant bits of the result. The most significant N bits of the input are called B and the least significant N bits are called C. By choosing appropriate values of B, C, and k, the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of A, *shamt*, and N for

(a) logical right shift of A by *shamt*

(b) arithmetic right shift of A by *shamt*

(c) left shift of A by *shamt*

(d) right rotate of A by *shamt*

(e) left rotate of A by *shamt*

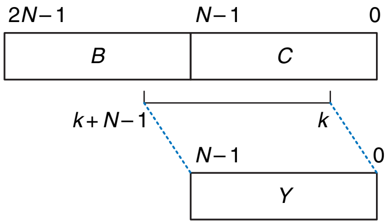


Figure 5.65 Funnel shifter

Exercise 5.22 Find the critical path for the 4 × 4 multiplier from Figure 5.20 in terms of an AND gate delay (t_{AND}) and an adder delay (t_{FA}) What is the delay of an $N \times N$ multiplier built in the same way?

Exercise 5.23 Find the critical path for the 4 × 4 divider from Figure 5.21 in terms of a 2:1 mux delay (t_{MUX}), an adder delay (t_{FA}), and an inverter delay (t_{INV}). What is the delay of an $N \times N$ divider built in the same way?

Exercise 5.24 Design a multiplier that handles two's complement numbers.

Exercise 5.25 A *sign extension unit* extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output (see Section 1.4.6). It receives an M-bit input A and produces an N-bit output Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.26 A *zero extension unit* extends an unsigned number from M to N bits ($N > M$) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.27 Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

Exercise 5.28 What is the range of numbers that can be represented by the following number systems?

(a) 24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits

(b) 24-bit sign and magnitude fixed-point numbers with 12 integer bits and 12 fraction bits

(c) 24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits

Exercise 5.29 Express the following base 10 numbers in 16-bit fixed-point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

(a) −13.5625

(b) 42.3125

(c) −17.15625

Exercise 5.30 Express the following base 10 numbers in 12-bit fixed-point sign/magnitude format with six integer bits and six fraction bits. Express your answer in hexadecimal.

(a) −30.5

(b) 16.25

(c) −8.078125

Exercise 5.31 Express the base 10 numbers in Exercise 5.29 in 16-bit fixed-point two's complement format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

Exercise 5.32 Express the base 10 numbers in Exercise 5.30 in 12-bit fixed-point two's complement format with six integer bits and six fraction bits. Express your answer in hexadecimal.

Exercise 5.33 Express the base 10 numbers in Exercise 5.29 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.34 Express the base 10 numbers in Exercise 5.30 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.35 Convert the following two's complement binary fixed-point numbers to base 10. The implied binary point is explicitly shown to aid in your interpretation.

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

Exercise 5.36 Repeat Exercise 5.35 for the following two's complement binary fixed-point numbers.

(a) 011101.10101

(b) 100110.11010

(c) 101000.00100

Exercise 5.37 When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

Exercise 5.38 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0123456 + 81C564B7

(b) D0B10301 + D1B43203

(c) 5EF10324 + 5E039020

Exercise 5.39 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0D20004 + 72407020

(b) C0D20004 + 40DC0004

(c) (5FBE4000 + 3FF80000) + DFDE4000
(Why is the result counterintuitive? Explain.)

Exercise 5.40 Expand the steps in section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

Exercise 5.41 Consider IEEE 754 single-precision floating-point numbers.

(a) How many numbers can be represented by IEEE 754 single-precision floating-point format? You need not count ±∞ or NaN.

(b) How many additional numbers could be represented if ±∞ and NaN were not represented?

(c) Explain why ±∞ and NaN are given special representations.

Exercise 5.42 Consider the following decimal numbers: 245 and 0.0625.

(a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.

(b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?

(c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.

(e) Does integer comparison work with your new floating-point notation from part (d)?

(f) Why is it convenient for integer comparison to work with floating-point numbers?

Exercise 5.43 Design a single-precision floating-point adder using your favorite HDL. Before coding the design in an HDL, sketch a schematic of your design. Simulate and test your adder to prove to a skeptic that it functions correctly. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

Exercise 5.44 In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

(a) Write the steps necessary to perform 32-bit floating-point multiplication.

(b) Sketch the schematic of a 32-bit floating-point multiplier.

(c) Design a 32-bit floating-point multiplier in an HDL. Simulate and test your multiplier to prove to a skeptic that it functions correctly.

Exercise 5.45 In this problem, you will explore the design of a 32-bit prefix adder.

(a) Sketch a schematic of your design.

(b) Design the 32-bit prefix adder in an HDL. Simulate and test your adder to prove that it functions correctly.

(c) What is the delay of your 32-bit prefix adder from part (a)? Assume that each two-input gate delay is 100 ps.

(d) Design a pipelined version of the 32-bit prefix adder. Sketch the schematic of your design. How fast can your pipelined prefix adder run? You may assume a sequencing overhead ($t_{pcq} + t_{\text{setup}}$) of 80 ps. Make the design run as fast as possible.

(e) Design the pipelined 32-bit prefix adder in an HDL.

Exercise 5.46 An incrementer adds 1 to an N-bit number. Build an 8-bit incrementer using half adders.

Exercise 5.47 Build a 32-bit synchronous *Up/Down counter*. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when $Up = 1$, the circuit counts up, and when $Up = 0$, the circuit counts down.

Exercise 5.48 Design a 32-bit counter that adds 4 at each clock edge. The counter has reset and clock inputs. Upon reset, the counter output is all 0.

Exercise 5.49 Modify the counter from Exercise 5.48 such that the counter will either increment by 4 or load a new 32-bit value, D, on each clock edge, depending on a control signal *Load*. When $Load = 1$, the counter loads the new value D.

Exercise 5.50 An N-bit *Johnson counter* consists of an N-bit shift register with a reset signal. The output of the shift register (S_{out}) is inverted and fed back to the input (S_{in}). When the counter is reset, all of the bits are cleared to 0.

(a) Show the sequence of outputs, $Q_{3:0}$, produced by a 4-bit Johnson counter starting immediately after the counter is reset.

(b) How many cycles elapse until an N-bit Johnson counter repeats its sequence? Explain.

(c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs $Y_{9:0}$. When the counter is reset, Y_0 is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.

(d) What advantages might a Johnson counter have over a conventional counter?

Exercise 5.51 Write the HDL for a 4-bit scannable flip-flop like the one shown in Figure 5.38. Simulate and test your HDL module to prove that it functions correctly.

Exercise 5.52 The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly a 1 will have at least three 1's.

(a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.

(b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.

(c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

Exercise 5.53 Flash EEPROM, simply called Flash memory, is a fairly recent invention that has revolutionized consumer electronics. Research and explain how Flash memory works. Use a diagram illustrating the floating gate. Describe how a bit in the memory is programmed. Properly cite your sources.

Exercise 5.54 The extraterrestrial life project team has just discovered aliens living on the bottom of Mono Lake. They need to construct a circuit to classify the aliens by potential planet of origin based on measured features available from the NASA probe: greenness, brownness, sliminess, and ugliness. Careful consultation with xenobiologists leads to the following conclusions:

- If the alien is green and slimy or ugly, brown, and slimy, it might be from Mars.
- If the critter is ugly, brown, and slimy, or green and neither ugly nor slimy, it might be from Venus.
- If the beastie is brown and neither ugly nor slimy or is green and slimy, it might be from Jupiter.

Note that this is an inexact science; for example, a life form which is mottled green and brown and is slimy but not ugly might be from either Mars or Jupiter.

(a) Program a 4 × 4 × 3 PLA to identify the alien. You may use dot notation.

(b) Program a 16 × 3 ROM to identify the alien. You may use dot notation.

(c) Implement your design in an HDL.

Exercise 5.55 Implement the following functions using a single 16 × 3 ROM. Use dot notation to indicate the ROM contents.

(a) $X = AB + B\overline{C}D + \overline{A}\,\overline{B}$

(b) $Y = AB + BD$

(c) $Z = A + B + C + D$

Exercise 5.56 Implement the functions from Exercise 5.55 using a 4 × 8 × 3 PLA. You may use dot notation.

Exercise 5.57 Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

(a) a 16-bit adder/subtractor with C_{in} and C_{out}

(b) an 8 × 8 multiplier

(c) a 16-bit priority encoder (see Exercise 2.36)

Exercise 5.58 Consider the ROM circuits in Figure 5.66. For each row, can the circuit in column I be replaced by an equivalent circuit in column II by proper programming of the latter's ROM?

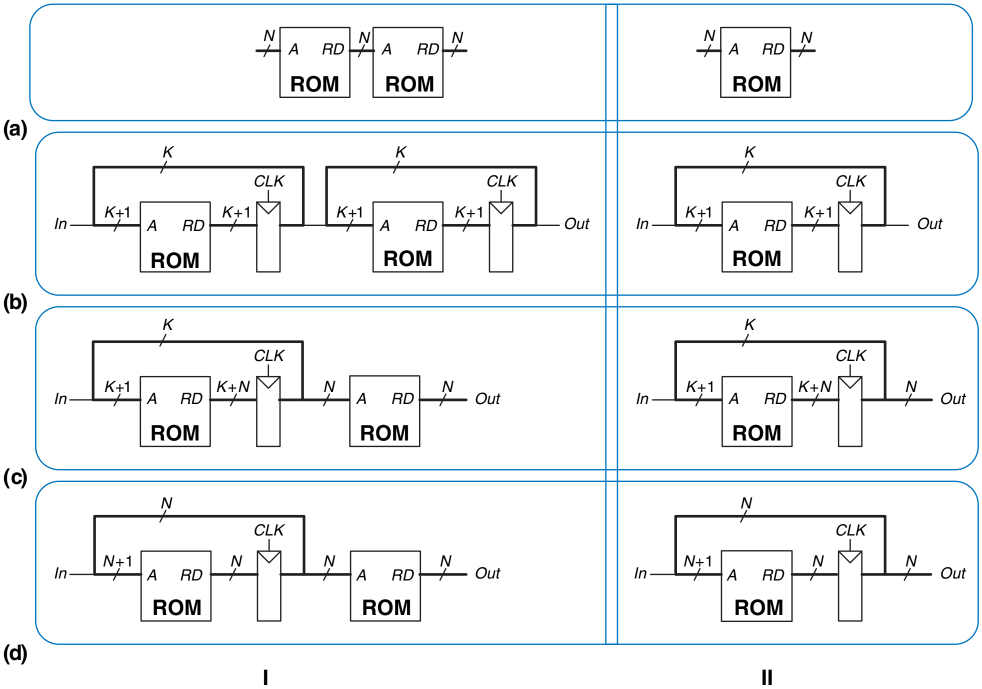


Figure 5.66 ROM circuits

Exercise 5.59 How many Cyclone IV FPGA LEs are required to perform each of the following functions? Show how to configure one or more LEs to perform the function. You should be able to do this by inspection, without performing logic synthesis.

(a) the combinational function from Exercise 2.13(c)

(b) the combinational function from Exercise 2.17(c)

(c) the two-output function from Exercise 2.24

(d) the function from Exercise 2.35

(e) a four-input priority encoder (see Exercise 2.36)

Exercise 5.60 Repeat Exercise 5.59 for the following functions.

(a) an eight-input priority encoder (see Exercise 2.36)

(b) a 3:8 decoder

(c) a 4-bit carry propagate adder (with no carry in or out)

(d) the FSM from Exercise 3.22

(e) the Gray code counter from Exercise 3.27

Exercise 5.61 Consider the Cyclone IV LE shown in Figure 5.59. According to the datasheet, it has the timing specifications given in Table 5.5.

(a) What is the minimum number of Cyclone IV LEs required to implement the FSM of Figure 3.26?

(b) Without clock skew, what is the fastest clock frequency at which this FSM will run reliably?

(c) With 3 ns of clock skew, what is the fastest frequency at which the FSM will run reliably?

Table 5.5 **Cyclone IV timing**

Name	Value (ps)
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Exercise 5.62 Repeat Exercise 5.61 for the FSM of Figure 3.31(b).

Exercise 5.63 You would like to use an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. The design is to be implemented as an FSM using a Cyclone IV FPGA. According to the data sheet, the FPGA has timing characteristics shown in Table 5.5. You would like your FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which the FSM will run?

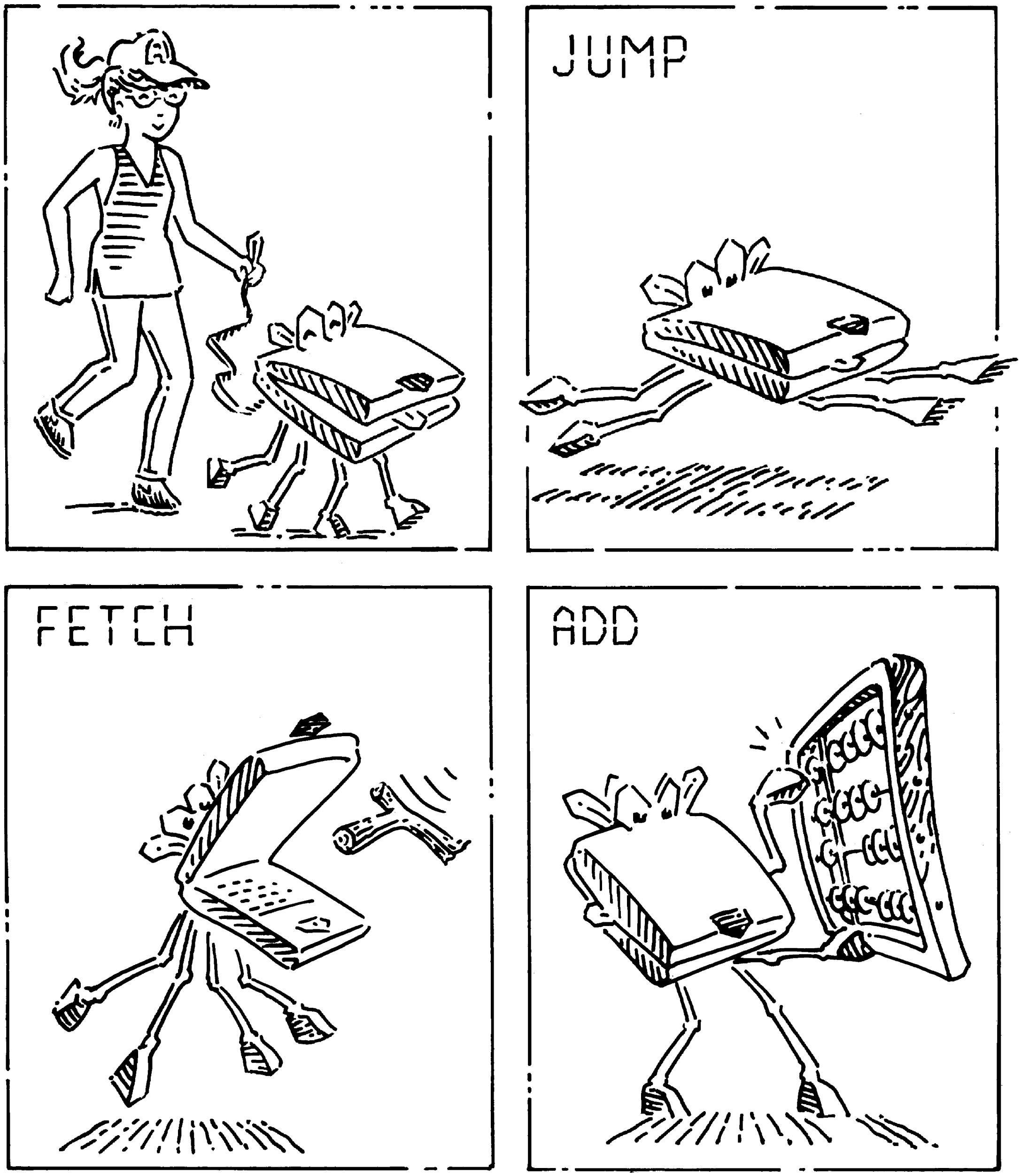
Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 5.1 What is the largest possible result of multiplying two unsigned N-bit numbers?

Question 5.2 *Binary coded decimal* (*BCD*) representation uses four bits to encode each decimal digit. For example, 42_{10} is represented as 01000010_{BCD}. Explain in words why processors might use BCD representation.

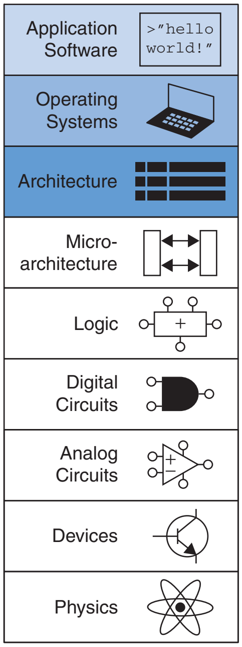
Question 5.3 Design hardware to add two 8-bit unsigned BCD numbers (see Question 5.2). Sketch a schematic for your design, and write an HDL module for the BCD adder. The inputs are A, B, and C_{in}, and the outputs are S and C_{out}. C_{in} and C_{out} are 1-bit carries and A, B, and S are 8-bit BCD numbers.


JUMP
FETCH
ADD

Architecture

6

6.1 Introduction
6.2 Assembly Language
6.3 Programming
6.4 Machine Language
6.5 Lights, Camera, Action: Compiling, Assembling, and Loading*
6.6 Odds and Ends*
6.7 Evolution of ARM Architecture
6.8 Another Perspective: x86 Architecture
6.9 Summary
Exercises
Interview Questions



6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and branch. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language. Just as we use letters to encode human language, computers use binary numbers to encode machine language. The ARM architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called assembly language.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and branch, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the microarchitecture and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

In this text, we introduce the ARM architecture. This architecture was first developed in the 1980s by Acorn Computer Group, which spun off Advanced RISC Machines Ltd., now known as ARM. Over 10 billion ARM processors are sold every year. Almost all cell phones and tablets contain multiple ARM processors. The architecture is used in everything from pinball machines to cameras to robots to cars to rack-mounted servers. ARM is unusual in that it does not sell processors directly, but rather licenses other companies to build its processors, often as part of a larger system-on-chip. For example, Samsung, Altera, Apple, and Qualcomm all build ARM processors, either using microarchitectures purchased from ARM or microarchitectures developed internally under license from ARM. We choose to focus on ARM because it is a commercial leader and because the architecture is clean, with few idiosyncrasies. We start by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

The "ARM architecture" we describe is ARM version 4 (ARMv4), which forms the core of the instruction set. Section 6.7 summarizes new features in versions 5–8 of the architecture. The ARM *Architecture Reference Manual* (ARM), available online, is the authoritative definition of the architecture.

Throughout the chapter, we motivate the design of the ARM architecture using four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design*: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the ARM instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java.

(These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

We used Keil's ARM Microcontroller Development Kit (MDK-ARM) to compile, assemble, and simulate the example assembly code in this chapter. The MDK-ARM is a free development tool that comes with a complete ARM compiler. Labs available on this textbook's companion site (see Preface) show how to install and use this tool to write, compile, simulate, and debug both C and assembly programs.

6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables `b` and `c` and writing the result to `a`. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in ARM assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	ARM Assembly Code
`a = b + c;`	`ADD a, b, c`

The first part of the assembly instruction, `ADD`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on `b` and `c`, the *source operands*, and the result is written to `a`, the *destination operand*.

Mnemonic (pronounced ni-mon-ik) comes from the Greek word *μιμνΕσκεστηαι*, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

Code Example 6.2 SUBTRACTION

High-Level Code	ARM Assembly Code
`a = b − c;`	`SUB a, b, c`

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the `ADD` instruction except for the operation specification, `SUB`. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple ARM instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with `//` and continue until the end of the line. Multiline comments begin with `/*` and end with `*/`. In ARM assembly language, only single-line comments

Code Example 6.3 MORE COMPLEX CODE

High-Level Code

```
a = b + c - d;    // single-line comment
                  /* multiple-line
                     comment */
```

ARM Assembly Code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

are used. They begin with a semicolon (`;`) and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable `t` to store the intermediate result. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The ARM instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, ARM is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a "string move" instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation. An instruction set with 256 complex instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1, the variables `a`, `b`, and `c` are all operands. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Version 8 of the ARM architecture has been extended to 64 bits, but we will focus on the 32-bit version in this book.

Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The ARM architecture uses 16 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small register file is faster than reading it from a large memory. A register file is typically built from a small SRAM array (see Section 5.5.3).

Code Example 6.4 shows the `ADD` instruction with register operands. ARM register names are preceded by the letter 'R'. The variables `a`, `b`, and `c` are arbitrarily placed in R0, R1, and R2. The name R1 is pronounced "register 1" or "R1" or "register R1". The instruction adds the 32-bit values contained in R1 (`b`) and R2 (`c`) and writes the 32-bit result to R0 (`a`). Code Example 6.5 shows ARM assembly code using a register, R4, to store the intermediate calculation of `b + c`:

Code Example 6.4 REGISTER OPERANDS

High-Level Code

```
a = b + c;
```

ARM Assembly Code

```
; R0 = a, R1 = b, R2 = c
  ADD R0, R1, R2        ; a = b + c
```

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code

```
a = b + c − d;
```

ARM Assembly Code

```
; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t
  ADD R4, R1, R2   ; t = b + c
  SUB R0, R4, R3   ; a = t − d
```

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into ARM assembly language. Assume variables `a–c` are held in registers R0–R2 and `f–j` are in R3–R7.

```
a = b − c;
f = (g + h) − (i + j);
```

Solution: The program uses four assembly language instructions.

```
; ARM assembly code
; R0 = a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j
  SUB R0, R1, R2     ; a = b − c
  ADD R8, R4, R5     ; R8 = g + h
  ADD R9, R6, R7     ; R9 = i + j
  SUB R3, R8, R9     ; f = (g + h) − (i + j)
```

The Register Set

Table 6.1 lists the name and use for each of the 16 ARM registers. R0–R12 are used for storing variables; R0–R3 also have special uses during procedure calls. R13–R15 are also called SP, LR, and PC, and they will be described later in this chapter.

Constants/Immediates

In addition to register operations, ARM instructions can use constant or *immediate* operands. These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the `ADD` instruction adding an immediate to a register. In assembly code, the immediate is preceded by the # symbol and can be written in decimal or hexadecimal. Hexadecimal constants in ARM assembly language start with 0x, as they

Table 6.1 **ARM register set**

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code

```
a = a + 4;
b = a − 12;
```

ARM Assembly Code

```
; R7 = a, R8 = b
  ADD R7, R7, #4     ; a = a + 4
  SUB R8, R7, #0xC   ; b = a − 12
```

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code

```
i = 0;
x = 4080;
```

ARM Assembly Code

```
; R4 = i, R5 = x
  MOV R4, #0       ; i = 0
  MOV R5, #0xFF0   ; x = 4080
```

do in C. Immediates are unsigned 8- to 12-bit numbers with a peculiar encoding described in Section 6.4.

The move instruction (`MOV`) is a useful way to initialize register values. Code Example 6.7 initializes the variables `i` and `x` to 0 and 4080, respectively. MOV can also take a register source operand. For example, `MOV R1, R7` copies the contents of register R7 into R1.

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 15 variables. However, data can also be stored in memory. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the ARM architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from Section 5.5 that memories are organized as an array of data words. The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in Figure 6.1(a). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4. The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Both the 32-bit word address and the data value in Figure 6.1(b) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

ARM provides the *load register* instruction, `LDR`, to read a data word from memory into a register. Code Example 6.8 loads memory word 2 into `a` (R7). In C, the number inside the brackets is the *index* or word number,

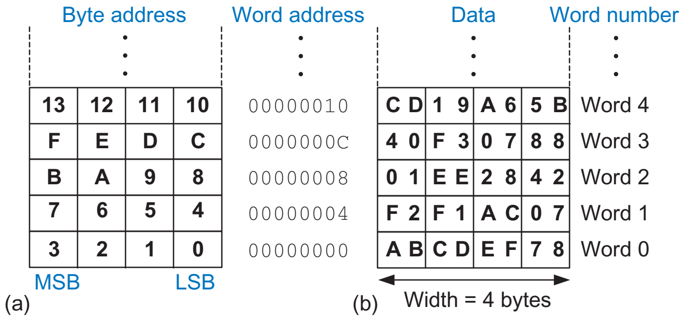


Figure 6.1 **ARM byte-addressable memory showing: (a) byte address and (b) data**

Code Example 6.8 READING MEMORY

High-Level Code

```
a = mem[2];
```

ARM Assembly Code

```
; R7 = a
  MOV R5, #0          ; base address = 0
  LDR R7, [R5, #8]    ; R7 <= data at memory address (R5+8)
```

A read from the base address (i.e., index 0) is a special case that requires no offset in the assembly code. For example, a memory read from the base address held in R5 is written as `LDR R3, [R5]`.

ARMv4 requires *word-aligned addresses* for `LDR` and `STR`, that is, a word address that is divisible by four. Since ARMv6, this alignment restriction can be removed by setting a bit in the ARM system control register, but performance of *unaligned* loads is usually worse. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others, such as MIPS, require strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `LDRB` and `STRB` (discussed in Section 6.3.6), need not be word aligned.

which we discuss further in Section 6.3.6. The `LDR` instruction specifies the memory address using a *base register* (R5) and an *offset* (8). Recall that each data word is 4 bytes, so word number 1 is at address 4, word number 2 is at address 8, and so on. The word address is four times the word number. The memory address is formed by adding the contents of the base register (R5) and the offset. ARM offers several modes for accessing memory, as will be discussed in Section 6.3.6.

After the load register instruction (`LDR`) is executed in Code Example 6.8, R7 holds the value 0x01EE2842, which is the data value stored at memory address 8 in Figure 6.1.

ARM uses the *store register* instruction, `STR`, to write a data word from a register into memory. Code Example 6.9 writes the value 42 from register R9 into memory word 5.

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.2. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word

Code Example 6.9 WRITING MEMORY

High-Level Code

```
mem[5] = 42;
```

ARM Assembly Code

```
MOV R1, #0          ; base address = 0
MOV R9, #42
STR R9, [R1, #0x14]  ; value stored at memory address (R1+20) = 42
```

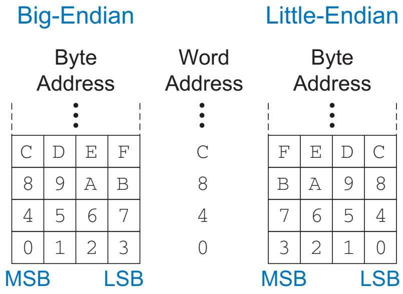
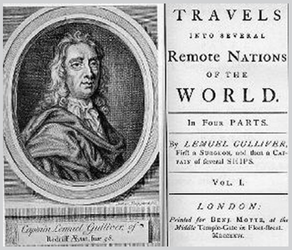


Figure 6.2 **Big-endian and little-endian memory addressing**

differ. In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. ARM prefers little-endian but provides support in some versions for *bi-endian* data addressing, which allows data loads and stores in either format. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.


TRAVELS
INTO SEVERAL
Remote NATIONS
OF THE
WORLD.
In Four PARTS.
By LEMUEL GULLIVER,
Firſt a Surgeon, and then a Captain of ſeveral SHIPS.
Vol. I.
LONDON:
Printed for Benj. Motte, at the Middle Temple-Gate in Fleet-ſtreet.
MDCCXXVI.

Captain Lemuel Gulliver, of Redriff

The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories, the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

These terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, Leeds University Library.)

6.3 PROGRAMMING

Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, conditional execution, if/else statements, for and while loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into ARM assembly code.

6.3.1 Data-processing Instructions

The ARM architecture defines a variety of *data-processing* instruction (often called logical and arithmetic instructions in other architectures). We introduce these instructions briefly here because they are necessary to implement higher-level constructs. Appendix B provides a summary of ARM instructions.

Logical Instructions

ARM *logical operations* include `AND`, `ORR` (OR), `EOR` (XOR), and `BIC` (bit clear). These each operate bitwise on two sources and write the result

Figure 6.3 **Logical operations**

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

to a destination register. The first source is always a register and the second source is either an immediate or another register. Another logical operation, `MVN` (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register. Figure 6.3 shows examples of these operations on the two source values 0x46A1F1B7 and 0xFFFF0000. The figure shows the values stored in the destination register after the instruction executes.

The bit clear (`BIC`) instruction is useful for masking bits (i.e., forcing unwanted bits to 0). `BIC R6, R1, R2` computes R1 AND NOT R2. In other words, `BIC` clears the bits that are asserted in R2. In this case, the top two bytes of R1 are cleared or *masked*, and the unmasked bottom two bytes of R1, 0xF1B7, are placed in R6. Any subset of register bits can be masked.

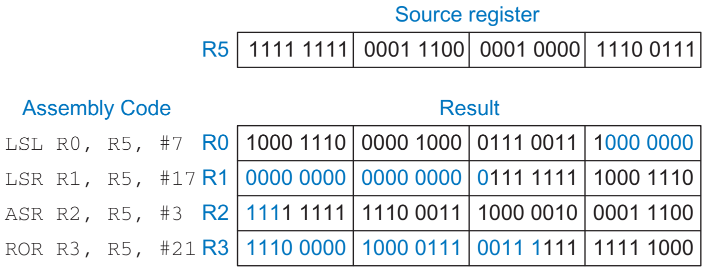
The `ORR` instruction is useful for combining bitfields from two registers. For example, 0x347A0000 `ORR` 0x000072FC = 0x347A72FC.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. The rotate instruction rotates the value in a register right by up to 31 bits. We refer to both shift and rotate generically as shift operations. ARM shift operations are `LSL` (logical shift left), `LSR` (logical shift right), `ASR` (arithmetic shift right), and `ROR` (rotate right). There is no `ROL` instruction because left rotation can be performed with a right rotation by a complementary amount.

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). The amount by which to shift can be an immediate or a register.

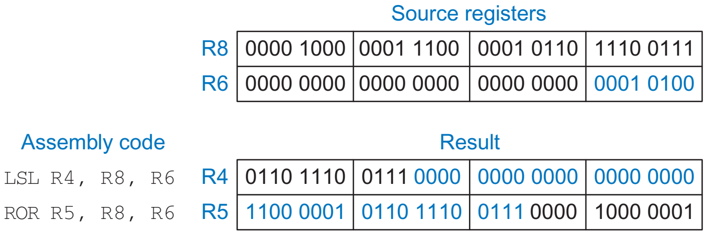
Figure 6.4 shows the assembly code and resulting register values for `LSL`, `LSR`, `ASR`, and `ROR` when shifting by an immediate value. R5 is shifted by the immediate amount, and the result is placed in the destination register.



Source register					
	R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Figure 6.4 **Shift instructions with immediate shift amounts**



Source registers					
	R8	0000 1000	0001 1100	0001 0110	1110 0111
	R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Figure 6.5 **Shift instructions with register shift amounts**

Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N, as discussed in Section 5.2.5. Logical shifts are also used to extract or assemble bitfields.

Figure 6.5 shows the assembly code and resulting register values for shift operations where the shift amount is held in a register, R6. This instruction uses the *register-shifted register* addressing mode, where one register (R8) is shifted by the amount (20) held in a second register (R6).

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. The ARM architecture provides *multiply instructions* that result in a 32-bit or 64-bit product. Multiply (`MUL`) multiplies two 32-bit numbers and produces a 32-bit result. `MUL R1, R2, R3` multiplies the values in R2 and R3 and places the least significant bits of the product in R1; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose result fits in 32 bits. `UMULL` (unsigned multiply long) and `SMULL` (signed multiply long) multiply two 32-bit numbers and produce a 64-bit product. For example, `UMULL R1, R2, R3, R4` performs an unsigned multiply of R3 and R4. The least significant 32 bits of the product is placed in R1 and the most significant 32 bits are placed in R2.

Each of these instructions also has a multiply-accumulate variant, `MLA`, `SMLAL`, and `UMLAL`, that adds the product to a running 32- or 64-bit sum. These instructions can boost the math performance in applications such as matrix multiplication and signal processing consisting of repeated multiplies and adds.

6.3.2 Condition Flags

Programs would be boring if they could only run in the same order every time. ARM instructions optionally set *condition flags* based on whether the result is negative, zero, etc. Subsequent instructions then execute *conditionally*, depending on the state of those condition flags. The ARM condition flags, also called *status flags*, are negative (*N*), zero (*Z*), carry (*C*), and overflow (*V*), as listed in Table 6.2. These flags are set by the ALU (see Section 5.2.4) and are held in the top 4 bits of the 32-bit *Current Program Status Register* (*CPSR*), as shown in Figure 6.6.

The most common way to set the status bits is with the compare (`CMP`) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the *Z* flag is set. If the first number is an unsigned value that is higher than or the same as the second, the subtraction will produce a carry out and the *C* flag is set.

Subsequent instructions can conditionally execute depending on the state of the flags. The instruction mnemonic is followed by a *condition mnemonic* that indicates when to execute. Table 6.3 lists the 4-bit condition field (*cond*), the condition mnemonic, name, and the state of the condition flags that result in instruction execution (CondEx). For example, suppose a program performs `CMP R4, R5`, and then `ADDEQ R1, R2, R3`. The compare sets the *Z* flag if R4 and R5 are equal, and the `ADDEQ` executes only if the *Z* flag is set. The *cond* field will be used in machine language encodings in Section 6.4.

Table 6.2 Condition flags

Flag	Name	Description
N	Negative	Instruction result is negative, i.e., bit 31 of the result is 1
Z	Zero	Instruction result is zero
C	Carry	Instruction causes a carry out
V	oVerflow	Instruction causes an overflow

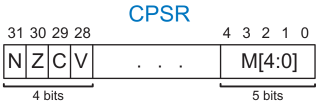


Figure 6.6 Current Program Status Register (CPSR)

The least significant five bits of the CPSR are *mode* bits and will be described in Section 6.6.3.

Other useful instructions for comparing two values are `CMN`, `TST`, and `TEQ`. Each instruction performs an operation, updates the condition flags, and discards the result. `CMN` (compare negative) compares the first source to the negative of the second source by adding the two sources. As will be shown in Section 6.4, ARM instructions only encode positive immediates. So, `CMN R2, #20` is used instead of `CMP R2, #-20`. `TST` (test) ANDs the source operands. It is useful for checking if some portion of the register is zero or nonzero. For example, `TST R2, #0xFF` would set the *Z* flag if the low byte of R2 is 0. `TEQ` (test if equal) checks for equivalence by XOR-ing the sources. Thus, the *Z* flag is set when they are equal and the *N* flag is set when the signs are different.

Table 6.3 **Condition mnemonics**

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Condition mnemonics differ for signed and unsigned comparison. For example, ARM provides two forms of greater than or equal comparison: HS (CS) is used for unsigned numbers and GE for signed. For unsigned numbers, $A - B$ will produce a carry out (C) when $A \geq B$. For signed numbers, $A - B$ will make N and V either both 0 or both 1 when $A \geq B$. Figure 6.7 highlights the difference between HS and GE comparisons with two examples using 4-bit numbers for ease of interpretation.

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, `SUBS R2, R3, R7` will subtract R7 from R3, put the result in R2, and set the condition flags. Table B.5 in Appendix B summarizes which condition flags are influenced by each instruction. All data-processing instructions will affect the N and Z flags based on whether the result is zero or has the most significant bit set. `ADDS` and `SUBS` also influence V and C, and shifts influence C.

Code Example 6.10 shows instructions that execute conditionally. The first instruction, `CMP R2, R3`, executes unconditionally and sets the condition flags. The remaining instructions execute conditionally, depending on the values of the condition flags. Suppose R2 and R3 contain the values 0x80000000 and 0x00000001. The compare computes R2 – R3 = 0x80000000 – 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF with a carry out ($C = 1$). The sources had opposite signs and the sign of the result differs from the sign of the first source, so the result overflows ($V = 1$). The remaining flags (N and Z) are 0. `ANDHS` executes

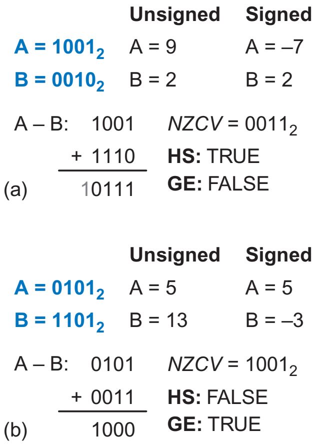


Figure 6.7 **Signed vs. unsigned comparison: HS vs. GE**

Code Example 6.10 CONDITIONAL EXECUTION

ARM Assembly Code

```
CMP    R2, R3
ADDEQ  R4, R5, #78
ANDHS  R7, R8, R9
ORRMI  R10, R11, R12
EORLT  R12, R7, R10
```

because $C = 1$. `EORLT` executes because N is 0 and V is 1 (see Table 6.3). Intuitively, `ANDHS` and `EORLT` execute because R2 ≥ R3 (unsigned) and R2 < R3 (signed), respectively. `ADDEQ` and `ORRMI` do not execute because the result of R2 – R3 is not zero (i.e., R2 ≠ R3) or negative.

6.3.3 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, if/else statements, switch/case statements, while loops, and for loops all conditionally execute code depending on some test.

One way to make decisions is to use conditional execution to ignore certain instructions. This works well for simple if statements where a small number of instructions are ignored, but it is wasteful for if statements with many instructions in the body, and it is insufficient to handle loops. Thus, ARM and most other architectures use *branch instructions* to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 after each instruction to point to the next instruction. (Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.) Branch instructions change the program counter. ARM includes two types of branches: a simple *branch* (`B`) and *branch and link* (`BL`). `BL` is used for function calls and is discussed in Section 6.3.7. Like other ARM instructions, branches can be unconditional or conditional. Branches are also called *jumps* in some architectures.

Code Example 6.11 shows unconditional branching using the branch instruction `B`. When the code reaches the `B TARGET` instruction, the branch is *taken*. That is, the next instruction executed is the `SUB` instruction just after the *label* called `TARGET`.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses (see Section 6.4.3). ARM assembly labels cannot be reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help

---

Code Example 6.11 UNCONDITIONAL BRANCHING

ARM Assembly Code

```
  ADD R1, R2, #17    ; R1 = R2 + 17
  B   TARGET         ; branch to TARGET
  ORR R1, R1, R3     ; not executed
  AND R3, R1, #0xFF  ; not executed

TARGET
  SUB R1, R1, #78    ; R1 = R1 − 78
```

Code Example 6.12 CONDITIONAL BRANCHING

ARM Assembly Code

```
  MOV R0, #4       ; R0 = 4
  ADD R1, R0, R0   ; R1 = R0 + R0 = 8
  CMP R0, R1       ; set flags based on R0−R1 = −4. NZCV = 1000
  BEQ THERE        ; branch not taken (Z != 1)
  ORR R1, R1, #1   ; R1 = R1 OR 1 = 9

THERE
  ADD R1, R1, #78  ; R1 = R1 + 78 = 87
```

make labels stand out. The ARM compiler makes this a requirement: labels must not be indented, and instructions must be preceded by white space. Some compilers, including GCC, require a colon after the label.

Branch instructions can execute conditionally based on the condition mnemonics listed in Table 6.3. Code Example 6.12 illustrates the use of `BEQ`, branching dependent on equality ($Z = 1$). When the code reaches the `BEQ` instruction, the Z condition flag is 0 (i.e., `R0` ≠ `R1`), so the branch is *not taken*. That is, the next instruction executed is the `ORR` instruction.

6.3.4 Conditional Statements

if, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into ARM assembly language.

if Statements

An if statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.13 shows how to translate an if statement into ARM assembly code.

Code Example 6.13 IF STATEMENT

High-Level Code

```
if (apples == oranges)
  f = i + 1;


f = f − i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP  R0, R1       ; apples == oranges ?
  BNE  L1           ; if not equal, skip if block
  ADD  R2, R3, #1   ; if block: f = i + 1
L1
  SUB  R2, R2, R3   ; f = f − i
```

Recall that != is an inequality comparison and == is an equality comparison in the high-level code.

The assembly code for the if statement tests the opposite condition of the one in the high-level code. In Code Example 6.13, the high-level code tests for `apples == oranges`. The assembly code tests for `apples != oranges` using `BNE` to skip the if block if the condition is **not** satisfied. Otherwise, `apples == oranges`, the branch is not taken, and the if block is executed.

Because any instruction can be conditionally executed, the ARM assembly code for Code Example 6.13 could also be written more compactly as shown below.

```
CMP   R0, R1          ; apples == oranges ?
ADDEQ R2, R3, #1      ; f = i + 1 on equality (i.e., Z = 1)
SUB   R2, R2, R3      ; f = f − i
```

This solution with conditional execution is shorter and also faster because it involves one fewer instruction. Moreover, we will see in Section 7.5.3 that branches sometimes introduce extra delay, whereas conditional execution is always fast. This example shows the power of conditional execution in the ARM architecture.

In general, when a block of code has a single instruction, it is better to use conditional execution rather than branch around it. As the block becomes longer, the branch becomes valuable because it avoids wasting time fetching instructions that will not be executed.

if/else Statements

if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.14 shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In Code Example 6.14, the high-level code tests for `apples == oranges`, and the assembly code tests for `apples != oranges`. If that opposite condition is TRUE, `BNE` skips the if block and executes the else block. Otherwise, the if block executes and finishes with an unconditional branch (`B`) past the else block.

Code Example 6.14 IF/ELSE STATEMENT

High-Level Code

```
if (apples == oranges)
  f = i + 1;



else
  f = f − i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1        ; apples == oranges?
  BNE L1            ; if not equal, skip if block
  ADD R2, R3, #1    ; if block: f = i + 1
  B   L2            ; skip else block
L1
  SUB R2, R2, R3    ; else block: f = f − i
L2
```

Again, because any instruction can conditionally execute and because the instructions within the if block do not change the condition flags, the ARM assembly code for Code Example 6.14 could also be written much more succinctly as:

```
CMP   R0, R1        ; apples == oranges?
ADDEQ R2, R3, #1    ; f = i + 1 on equality (i.e., Z = 1)
SUBNE R2, R2, R3    ; f = f − i on not equal (i.e., Z = 0)
```

switch/case Statements*

switch/case statements execute one of several blocks of code depending on the conditions. If no conditions are met, the *default block* is executed. A case statement is equivalent to a series of *nested* if/else statements. Code Example 6.15 shows two high-level code snippets with the same

Code Example 6.15 SWITCH/CASE STATEMENT

High-Level Code

```
switch (button) {
  case 1:  amt = 20;  break;




  case 2:  amt = 50;  break;




  case 3:  amt = 100; break;



  default: amt = 0;
}
// equivalent function using
// if/else statements
  if      (button == 1) amt = 20;
  else if (button == 2) amt = 50;
  else if (button == 3) amt = 100;
  else                  amt = 0;
```

ARM Assembly Code

```
; R0 = button, R1 = amt
  CMP   R0, #1       ; is button 1 ?
  MOVEQ R1, #20      ; amt = 20 if button is 1
  BEQ   DONE         ; break

  CMP   R0, #2       ; is button 2 ?
  MOVEQ R1, #50      ; amt = 50 if button is 2
  BEQ   DONE         ; break

  CMP   R0, #3       ; is button 3?
  MOVEQ R1, #100     ; amt = 100 if button is 3
  BEQ   DONE         ; break

  MOV   R1, #0       ; default amt = 0
DONE
```

functionality: they calculate whether to dispense \$20, \$50, or \$100 from an ATM (automatic teller machine) depending on the button pressed. The ARM assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. while loops and for loops are common loop constructs used by high-level languages. This section shows how to translate them into ARM assembly language, taking advantage of conditional branching.

while Loops

while loops repeatedly execute a block of code until a condition is *not* met. The while loop in Code Example 6.16 determines the value of `x` such that $2^x = 128$. It executes seven times, until `pow` = 128.

Like if/else statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, R0 == 128), the while loop is finished. If not (R0 ≠128), the branch isn't taken and the loop body executes.

The `int` data type in C refers to a word of data representing a two's complement integer. ARM uses 32-bit words, so an `int` represents a number in the range $[-2^{31}, 2^{31} - 1]$.

Code Example 6.16 WHILE LOOP

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1      ; pow = 1
  MOV R1, #0      ; x = 0

WHILE
  CMP R0, #128    ; pow != 128 ?
  BEQ DONE        ; if pow == 128, exit loop
  LSL R0, R0, #1  ; pow = pow * 2
  ADD R1, R1, #1  ; x = x + 1
  B   WHILE       ; repeat loop
DONE
```

In Code Example 6.16, the while loop compares `pow` to 128 and exits the loop if it is equal. Otherwise it doubles `pow` (using a left shift), increments `x`, and branches back to the start of the while loop.

for Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time through the while loop. for loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The format of the for loop is:

```
for (initialization; condition; loop operation)
  statement
```

Code Example 6.17 FOR LOOP

High-Level Code

```
int i;
int sum = 0;



for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
}
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0        ; sum = 0
  MOV R0, #0        ; i = 0          loop initialization

FOR
  CMP R0, #10       ; i < 10 ?       check condition
  BGE DONE          ; if (i >= 10) exit loop
  ADD R1, R1, R0    ; sum = sum + i  loop body
  ADD R0, R0, #1    ; i = i + 1      loop operation
  B   FOR           ; repeat loop
DONE
```

The initialization code executes before the for loop begins. The condition is tested at the beginning of each loop. If the condition is not met, the loop exits. The loop operation executes at the end of each loop.

Code Example 6.17 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as `i` is less than 10. Note that this example also illustrates relative comparisons. The loop checks the < condition to continue, so the assembly code checks the opposite condition, >=, to exit the loop.

Loops are especially useful for accessing large amounts of similar data stored in memory, which is discussed next.

6.3.6 Memory

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array.

Figure 6.8 shows a 200-element array of scores stored in memory. Code Example 6.18 is a grade inflation algorithm that adds 10 points to each of the scores. Note that the code for initializing the scores array is not shown. The index into the array is a variable (`i`) rather than a constant, so we must multiply it by 4 before adding it to the base address.

ARM can *scale* (multiply) the index, add it to the base address, and load from memory in a single instruction. Instead of the `LSL` and `LDR` instruction sequence in Code Example 6.18, we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

R1 is scaled (shifted left by two) then added to the base address (R0). Thus, the memory address is R0 + (R1 × 4).

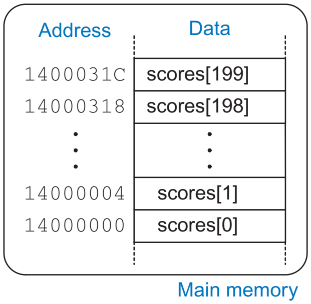


Figure 6.8 **Memory holding** `scores[200]` **starting at base address 0x14000000**

Code Example 6.18 ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...



for (i = 0; i < 200; i = i + 1)



   scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code       ...
  MOV R0, #0x14000000       ; R0 = base address
  MOV R1, #0                ; i = 0

LOOP
  CMP R1, #200              ; i < 200?
  BGE L3                    ; if i ≥ 200, exit loop
  LSL R2, R1, #2            ; R2 = i * 4
  LDR R3, [R0, R2]          ; R3 = scores[i]
  ADD R3, R3, #10           ; R3 = scores[i] + 10
  STR R3, [R0, R2]          ; scores[i] = scores[i] + 10
  ADD R1, R1, #1            ; i = i + 1
  B   LOOP                  ; repeat loop
L3
```

In addition to scaling the index register, ARM provides offset, pre-indexed, and post-indexed addressing to enable dense and efficient code for array accesses and function calls. Table 6.4 gives examples of each indexing mode. In each case, the base register is R1 and the offset is R2. The offset can be subtracted by writing –R2. The offset may also be an immediate in the range of 0–4095 that can be added (e.g., #20) or subtracted (e.g., #–20).

Offset addressing calculates the address as the base register ± the offset; the base register is unchanged. *Pre-indexed addressing* calculates the address as the base register ± the offset and updates the base register to this new address. *Post-indexed addressing* calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register ± the offset. We have seen many examples of offset indexing mode. Code Example 6.19 shows the for loop from Code Example 6.18 rewritten to use post-indexing, eliminating the `ADD` to increment `i`.

Table 6.4 **ARM indexing modes**

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Code Example 6.19 FOR LOOP USING POST-INDEXING

High-Level Code

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code   ...
  MOV R0, #0x14000000    ; R0 = base address
  ADD R1, R0, #800       ; R1 = base address + (200*4)

LOOP
  CMP R0, R1             ; reached end of array?
  BGE L3                 ; if yes, exit loop
  LDR R2, [R0]           ; R2 = scores[i]
  ADD R2, R2, #10        ; R2 = scores[i] + 10
  STR R2, [R0], #4       ; scores[i] = scores[i] + 10
                         ; then R0 = R0 + 4
  B   LOOP               ; repeat loop
 L3
```

Bytes and Characters

Numbers in the range [−128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.5 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see www.unicode.org.

ARM provides load byte (`LDRB`), load signed byte (`LDRSB`), and store byte (`STRB`) to access individual bytes in memory. `LDRB` zero-extends the byte, whereas `LDRSB` sign-extends the byte to fill the entire 32-bit register. `STRB` stores the least significant byte of the 32-bit register into the specified byte address in memory. All three are illustrated in Figure 6.9, with

`LDRH`, `LDRSH`, and `STRH` are similar, but access 16-bit *halfwords*.

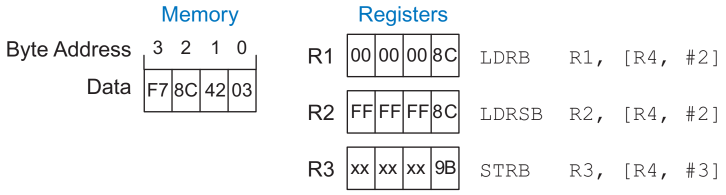


Figure 6.9 Instructions for loading and storing bytes

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as – , – … , – . – . , and – .. , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001.

However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

Table 6.5 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

the base address R4 being 0. `LDRB` loads the byte at memory address 2 into the least significant byte of R1 and fills the remaining register bits with 0. `LDRSB` loads this byte into R2 and sign-extends the byte into the upper 24 bits of the register. `STRB` stores the least significant byte of R3 (0x9B) into memory byte 3; it replaces 0xF7 with 0x9B. The more significant bytes of R3 are ignored.

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.10 shows the string “Hello!” (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long

Example 6.2 USING LDRB AND STRB TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into ARM assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that R0 already holds the base address of `chararray`.

```
// high-level code
// chararray[10] declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] − 32;
```

Solution:

```
; ARM assembly code
; R0 = base address of chararray (initialized earlier), R1 = i
          MOV   R1, #0            ; i = 0
LOOP      CMP   R1, #10           ; i < 10 ?
          BGE   DONE              ; if (i >=10), exit loop
          LDRB  R2, [R0, R1]      ; R2 = mem[R0+R1] = chararray[i]
          SUB   R2, R2, #32       ; R2 = chararray[i] − 32
          STRB  R2, [R0, R1]      ; chararray[i] = R2
          ADD   R1, R1, #1        ; i = i + 1
          B     LOOP              ; repeat loop
DONE
```

and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

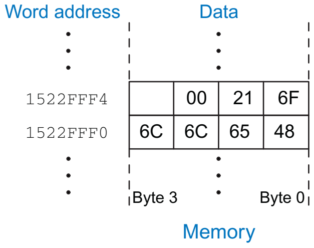


Figure 6.10 The string "Hello!" stored in memory

6.3.7 Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call, and the callee places the return value in register R0 before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the link register LR at the same time it jumps to the callee using the branch and link instruction (`BL`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (R4–R11, and LR) and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It shows how functions access input arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

ARM uses the branch and link instruction (`BL`) to call a function and moves the link register to the PC (`MOV PC, LR`) to return from a function. Code Example 6.20 shows the `main` function calling the `simple` function. `main` is the caller, and `simple` is the callee. The `simple` function is called with no input arguments and generates no return value; it just returns to the caller. In Code Example 6.20, instruction addresses are given to the left of each ARM instruction in hexadecimal.

`BL` (branch and link) and `MOV PC, LR` are the two essential instructions needed for a function call and return. `BL` performs two tasks: it stores the *return address* of the next instruction (the instruction

Code Example 6.20 simple FUNCTION CALL

High-Level Code

```
int main() {

  simple();
  ...
}

// void means the function returns no value
void simple() {
  return;
}
```

ARM Assembly Code

```
0x00008000 MAIN      ...
...                  ...
0x00008020           BL   SIMPLE    ; call the simple function
...


0x0000902C  SIMPLE   MOV PC, LR     ; return
```

after `BL`) in the link register (LR), and it branches to the target instruction.

In Code Example 6.20, the `main` function calls the `simple` function by executing the branch and link instruction (`BL`). `BL` branches to the `SIMPLE` label and stores 0x00008024 in LR. The `simple` function returns immediately by executing the instruction `MOV PC, LR`, copying the return address from the LR back to the PC. The `main` function then continues executing at this address (0x00008024).

Input Arguments and Return Values

The `simple` function in Code Example 6.20 receives no input from the calling function (`main`) and returns no output. By ARM convention, functions use R0–R3 for input arguments and R0 for the return value. In Code Example 6.21, the function `diffofsums` is called with four arguments and returns one result. `result` is a local variable, which we choose to keep in R4.

According to ARM convention, the calling function, `main`, places the function arguments from left to right into the input registers, R0–R3. The called function, `diffofsums`, stores the return value in the return register, R0. When a function with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.21 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main() {
  int y;
  . . .
  y = diffofsums(2, 3, 4, 5);
  . . .
}

int diffofsums(int f, int g, int h, int i) {
  int result;

  result = (f + g) − (h + i);
  return result;
}
```

ARM Assembly Code

```
; R4 = y
MAIN
  . . .
  MOV R0, #2       ; argument 0 = 2
  MOV R1, #3       ; argument 1 = 3
  MOV R2, #4       ; argument 2 = 4
  MOV R3, #5       ; argument 3 = 5
  BL  DIFFOFSUMS   ; call function
  MOV R4, R0       ; y = returned value
  . . .

; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1   ; R8 = f + g
  ADD R9, R2, R3   ; R9 = h + i
  SUB R4, R8, R9   ; result = (f + g) − (h + i)
  MOV R0, R4       ; put return value in R0
  MOV PC, LR       ; return to caller
```

Remember that PC and LR are alternative names for R15 and R14, respectively. ARM is unusual in that PC is part of the register set, so a function return can be done with a `MOV` instruction. Many other instruction sets keep the PC in a special register and use a special return or jump instruction to return from functions.

These days, ARM compilers do a function return using `BX LR`. The `BX` branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set described in Section 6.7.1. This chapter doesn't use the Thumb or `BX` instructions and thus sticks with the ARMv4 `MOV PC, LR` method.

We will see in Chapter 7 that treating the PC as an ordinary register complicates the implementation of the processor.

Code Example 6.21 has some subtle errors. Code Examples 6.22–6.25 show improved versions of the program.

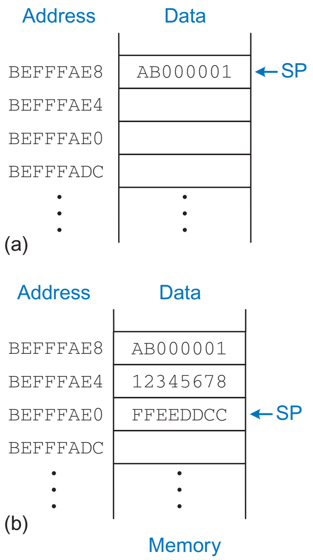


Figure 6.11 **The stack (a) before expansion and (b) after two-word expansion**

The Stack

The stack is memory that is used to save information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary values, we explain how the stack works.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be *popped* off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the ARM stack grows down in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.11 shows a picture of the stack. The stack pointer, SP (R13), is an ordinary ARM register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. SP points to (gives the address of) data. For example, in Figure 6.11(a), the stack pointer, SP, holds the address value 0XBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (SP) starts at a high memory address and decrements to expand as needed. Figure 6.11(b) shows the stack expanding to allow two more data words of temporary storage. To do so, SP decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing the return value. The `diffofsums` function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9. If `main` had been using these registers before the call to `diffofsums`, their contents would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. ARM also allows for *ascending stacks* that grow up toward higher memory addresses. The stack pointer typically points to the topmost element on the stack; this is called a *full stack*. ARM also allows for *empty stacks* in which SP points one word beyond the top of the stack. The ARM *Application Binary Interface* (ABI) defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate. It specifies a *full descending* stack, which we will use in this chapter.

Code Example 6.22 shows an improved version of `diffofsums` that saves and restores R4, R8, and R9. Figure 6.12 shows the stack before, during, and after a call to the `diffofsums` function from Code Example 6.22. The stack starts at 0xBEF0F0FC. `diffofsums` makes room for three words on the stack by decrementing the stack pointer SP by 12. It then stores the current values held in R4, R8, and R9 in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, R0 holds the result, but there

Code Example 6.22 FUNCTION SAVING REGISTERS ON THE STACK

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
  SUB   SP, SP, #12       ; make space on stack for 3 registers
  STR   R9, [SP, #8]      ; save R9 on stack
  STR   R8, [SP, #4]      ; save R8 on stack
  STR   R4, [SP]          ; save R4 on stack

  ADD  R8, R0, R1         ; R8 = f + g
  ADD  R9, R2, R3         ; R9 = h + i
  SUB  R4, R8, R9         ; result = (f + g) − (h + i)
  MOV  R0, R4             ; put return value in R0

  LDR   R4, [SP]          ; restore R4 from stack
  LDR   R8, [SP, #4]      ; restore R8 from stack
  LDR   R9, [SP, #8]      ; restore R9 from stack
  ADD   SP, SP, #12       ; deallocate stack space

  MOV   PC, LR            ; return to caller
```

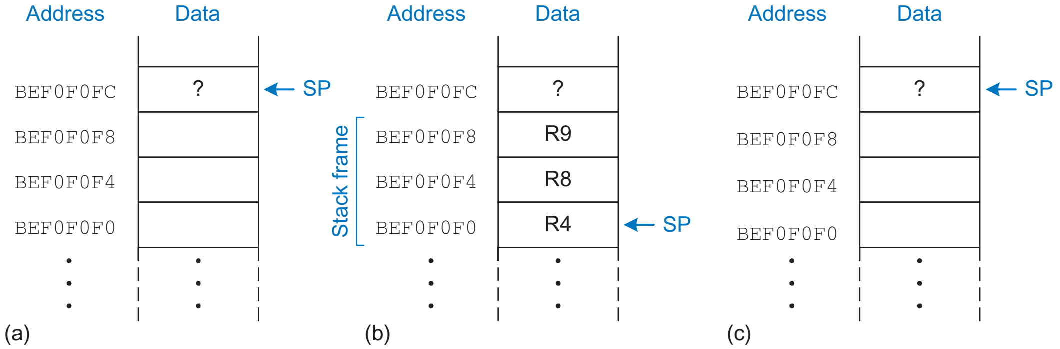

Address
Data
BEF0F0FC
?
SP
BEF0F0F8
BEF0F0F4
BEF0F0F0
(a)
Address
Data
Stack frame
BEF0F0FC
?
BEF0F0F8
R9
BEF0F0F4
R8
BEF0F0F0
R4
SP
(b)
Address
Data
BEF0F0FC
?
SP
BEF0F0F8
BEF0F0F4
BEF0F0F0
(c)

Figure 6.12 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

are no other side effects: R4, R8, R9, and SP have the same values as they did before the function call.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (`LDM` and `STM`) that are optimized to this purpose. Code Example 6.23 rewrites `diffofsums` using these instructions. The stack holds exactly the same information as in the previous example, but the code is much shorter.

Code Example 6.23 SAVING AND RESTORING MULTIPLE REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  STMFD SP!, {R4, R8, R9}       ; push R4/8/9 on full descending stack

  ADD   R8, R0, R1              ; R8 = f + g
  ADD   R9, R2, R3              ; R9 = h + i
  SUB   R4, R8, R9              ; result = (f + g) − (h + i)
  MOV   R0, R4                  ; put return value in R0

  LDMFD SP!, {R4, R8, R9}       ; pop R4/8/9 off full descending stack

  MOV   PC, LR                  ; return to caller
```

`LDM` and `STM` come in four flavors for full and empty descending and ascending stacks (`FD`, `ED`, `FA`, `EA`). The `SP!` in the instructions indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load. `PUSH` and `POP` are synonyms for `STMFD SP!, {regs}` and `LDMFD SP!, {regs}`, respectively, and are the preferred way to save registers on the conventional full descending stack.

Preserved Registers

Code Examples 6.22 and 6.23 assume that all of the used registers (R4, R8, and R9) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, ARM divides registers into *preserved* and *nonpreserved* categories. The preserved registers include R4–R11. The nonpreserved registers are R0–R3 and R12. SP and LR (R13 and R14) must also be preserved. A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.24 shows a further improved version of `diffofsums` that saves only R4 on the stack. It also illustrates the preferred `PUSH` and `POP` synonyms. The code reuses the nonpreserved argument registers R1 and R3 to hold the intermediate sums when those arguments are no longer necessary.

Code Example 6.24 REDUCING THE NUMBER OF PRESERVED REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  PUSH {R4}                       ; save R4 on stack
  ADD  R1, R0, R1                 ; R1 = f + g
  ADD  R3, R2, R3                 ; R3 = h + i
  SUB  R4, R1, R3                 ; result = (f + g) − (h + i)
  MOV  R0, R4                     ; put return value in R0
  POP  {R4}                       ; pop R4 off stack
  MOV  PC, LR                     ; return to caller
```

`PUSH` (and `POP`) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction. For example, `PUSH {R8, R1, R3}` will store R1 at the lowest memory address, then R3 and finally R8 at the next higher memory addresses on the stack.

Remember that when one function calls another, the former is the caller and the latter is the callee. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Table 6.6 summarizes which registers are preserved. R4–R11 are generally used to hold local variables within a function, so they must be saved. LR must also be saved, so that the function knows where to return.

Table 6.6 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

The convention of which registers are preserved or not preserved is part of the Procedure Call Standard for the ARM Architecture, rather than of the architecture itself. Alternate procedure call standards exist.

R0–R3 and R12 are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them.

R0–R3 are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. R0 certainly should not be preserved, because the callee returns its result in this register. Recall that the Current Program Status Register (CPSR) holds the condition flags. It is not preserved across function calls.

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above SP. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from SP at the beginning of the function.

The astute reader or an optimizing compiler may notice that the local variable `result` is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store it in the return register R0, eliminating the need to push and pop R4 and to move `result` from R4 to R0. Code Example 6.25 shows this even further optimized `diffofsums`.

Nonleaf Function Calls

A function that does not call others is called a *leaf function*; `diffofsums` is an example. A function that does call others is called a *nonleaf function*. As mentioned, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically:

Caller save rule: Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.

Code Example 6.25 OPTIMIZED `diffofsums` FUNCTION CALL

ARM Assembly Code

```
DIFFOFSUMS
  ADD   R1, R0, R1     ; R1 = f + g
  ADD   R3, R2, R3     ; R3 = h + i
  SUB   R0, R1, R3     ; return (f + g) − (h + i)
  MOV   PC, LR         ; return to caller
```

Code Example 6.26 demonstrates a nonleaf function `f1` and a leaf function `f2` including all the necessary saving and preserving of registers. Suppose `f1` keeps `i` in R4 and `x` in R5. `f2` keeps `r` in R4. `f1` uses preserved registers R4, R5, and LR, so it initially pushes them on the stack according to the callee save rule. It uses R12 to hold the intermediate result `(a – b)` so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` pushes R0 and R1 onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. Although R12 is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs R12 and doesn't have to save it. `f1` then passes the argument to `f2` in R0, makes the function call, and uses the result in R0. `f1` then restores R0 and R1 because it still needs them. When `f1` is done, it puts the return value in R0, restores preserved registers R4, R5, and LR, and returns. `f2` saves and restores R4 according to the callee save rule.

A nonleaf function overwrites LR when it calls another function using `BL`. Thus, a nonleaf function must always save LR on its stack and restore it before returning.

Code Example 6.26 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a − b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
```

```
int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
  PUSH {R4, R5, LR}   ; save preserved registers used by f1
  ADD  R5, R0, R1     ; x = (a + b)
  SUB  R12, R0, R1    ; temp = (a − b)
  MUL  R5, R5, R12    ; x = x * temp = (a + b) * (a − b)
  MOV  R4, #0         ; i = 0
FOR
  CMP  R4, R0         ; i < a?
  BGE  RETURN         ; no: exit loop
  PUSH {R0, R1}       ; save nonpreserved registers
  ADD  R0, R1, R4     ; argument is b + i
  BL   F2             ; call f2(b+i)
  ADD  R5, R5, R0     ; x = x + f2(b+i)
  POP  {R0, R1}       ; restore nonpreserved registers
  ADD  R4, R4, #1     ; i++
  B    FOR            ; continue for loop
RETURN
  MOV  R0, R5         ; return value is x
  POP  {R4, R5, LR}   ; restore preserved registers
  MOV  PC, LR         ; return from f1
```

```
; R0 = p, R4 = r
F2
  PUSH {R4}           ; save preserved registers used by f2
  ADD  R4, R0, 5      ; r = p + 5
  ADD  R0, R4, R0     ; return value is r + p
  POP  {R4}           ; restore preserved registers
  MOV  PC, LR         ; return from f2
```

On careful inspection, one might note that `f2` does not modify R1, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call.

An optimizing compiler could observe that `f2` is a leaf procedure and could allocate `r` to a nonpreserved register, avoiding the need to save and restore R4.

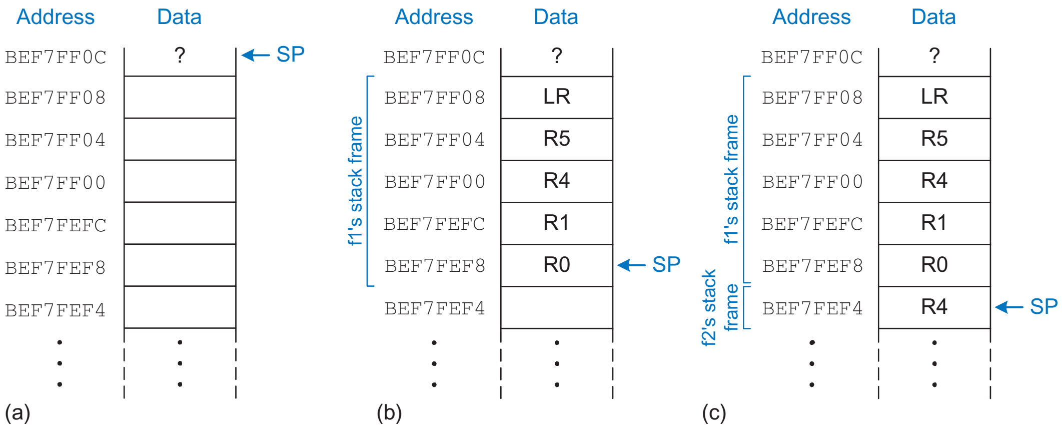


Figure 6.13 **The stack: (a) before function calls, (b) during `f1`, and (c) during `f2`**

Figure 6.13 shows the stack during execution of `f1`. The stack pointer originally starts at 0xBEF7FF0C.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. The factorial function can be rewritten recursively as $factorial(n) = n \times factorial(n-1)$, as shown in Code Example 6.27. The factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address 0x8500.

According to the callee save rule, `factorial` is a nonleaf function and must save LR. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save R0. Hence, it pushes both registers onto the stack at the start. It then checks whether `n` ≤ 1. If so, it puts the return value of 1 in R0, restores the stack pointer, and returns to the caller. It does not have to reload LR and R0 in this case, because they were never modified. If `n` > 1, the function recursively calls `factorial(n - 1)`. It then restores the value of `n` and the link register (LR) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into R1, so as not to overwrite the returned value. The multiply instruction (`MUL R0, R1, R0`) multiplies `n` (R1) and the returned value (R0) and puts the result in R0.

Code Example 6.27 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;



  else
    return (n * factorial(n - 1));
}
```

ARM Assembly Code

```
0x8500  FACTORIAL  PUSH  {R0, LR}    ; push n and LR on stack
0x8504             CMP   R0, #1      ; R0 <= 1?
0x8508             BGT   ELSE        ; no: branch to else
0x850C             MOV   R0, #1      ; otherwise, return 1
0x8510             ADD   SP, SP, #8  ; restore SP
0x8514             MOV   PC, LR      ; return
0x8518  ELSE       SUB   R0, R0, #1  ; n = n - 1
0x851C             BL    FACTORIAL   ; recursive call
0x8520             POP   {R1, LR}    ; pop n (into R1) and LR
0x8524             MUL   R0, R1, R0  ; R0 = n * factorial(n - 1)
0x8528             MOV   PC, LR      ; return
```

Figure 6.14 shows the stack when executing `factorial(3)`. For illustration, we show SP initially pointing to 0xBEFF0FF0, as shown in Figure 6.14(a). The function creates a two-word stack frame to hold `n` (R0) and LR. On the first invocation, `factorial` saves R0 (holding `n` = 3) at 0xBEFF0FE8 and LR at 0xBEFF0FEC, as shown in Figure 6.14(b). The function then changes `n` to 2 and recursively calls `factorial(2)`, making LR hold 0x8520. On the second invocation, it saves R0 (holding n=2) at 0xBEFF0FE0 and LR at 0xBEFF0FE4. This time, we know that LR contains 0x8520. The function then changes `n` to 1 and recursively calls `factorial(1)`. On the third invocation, it saves R0 (holding `n` = 1) at

For clarity, we will always save registers at the start of a procedure call. An optimizing compiler might observe that there is no need to save R0 and LR when `n` ≤ 1, and thus push registers only in the `ELSE` portion of the function.

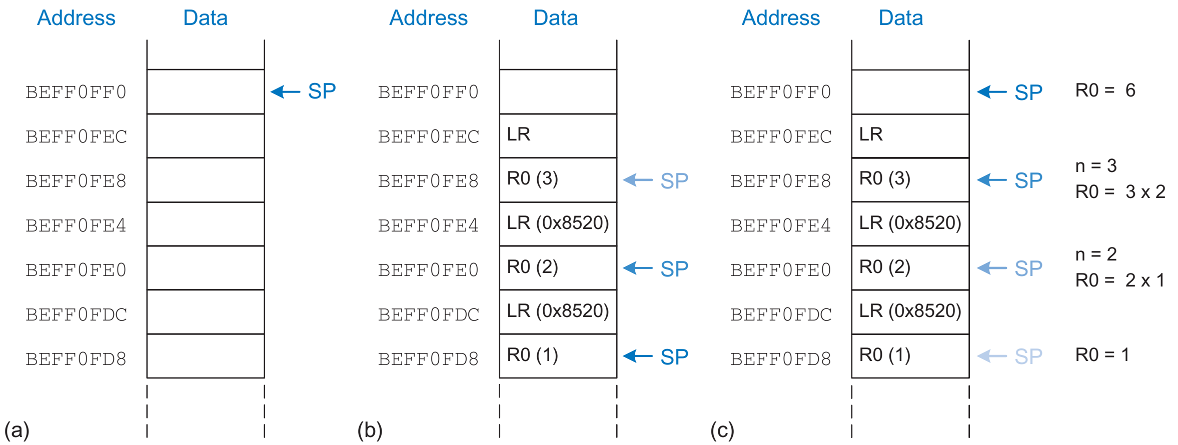

Address
Data
BEFF0FF0
BEFF0FEC
BEFF0FE8
BEFF0FE4
BEFF0FE0
BEFF0FDC
BEFF0FD8
SP
(a)
Address
Data
BEFF0FF0
BEFF0FEC
BEFF0FE8
BEFF0FE4
BEFF0FE0
BEFF0FDC
BEFF0FD8
LR
R0 (3)
LR (0x8520)
R0 (2)
LR (0x8520)
R0 (1)
SP
SP
SP
(b)
Address
Data
BEFF0FF0
BEFF0FEC
BEFF0FE8
BEFF0FE4
BEFF0FE0
BEFF0FDC
BEFF0FD8
LR
R0 (3)
LR (0x8520)
R0 (2)
LR (0x8520)
R0 (1)
SP
SP
SP
SP
R0 = 6
n = 3
R0 = 3 x 2
n = 2
R0 = 2 x 1
R0 = 1
(c)

Figure 6.14 **Stack: (a) before, (b) during, and (c) after factorial function call with** `n` **= 3**

0xBEFF0FD8 and LR at 0xBEFF0FDC. This time, LR again contains 0x8520. The third invocation of `factorial` returns the value 1 in R0 and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` (into R1) to 2, restores LR to 0x8520 (it happened to already have this value), deallocates the stack frame, and returns R0 = 2 × 1 = 2 to the first invocation. The first invocation restores `n` (into R1) to 3, restores LR to the return address of the caller, deallocates the stack frame, and returns R0 = 3 × 2 = 6. Figure 6.14(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xBEFF0FF0), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. R0 holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers. The stack is used to store this information. By ARM convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP. The caller must expand its stack to make room for the additional arguments. Figure 6.15(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in R4–R11; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.15(b) shows the organization of a callee's stack frame. The stack frame holds the temporary registers and link register (if they need to be saved because of a subsequent function call), and any of the saved

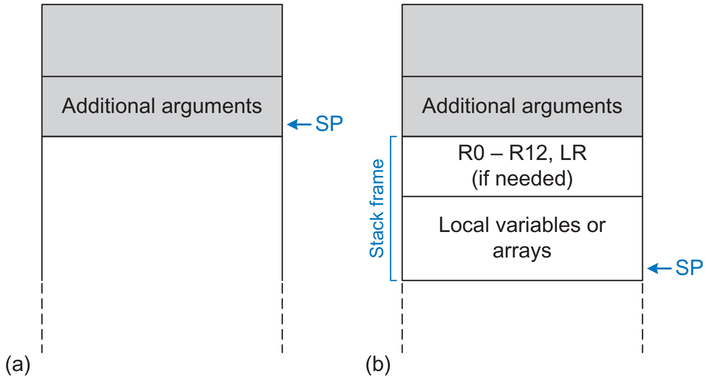


Figure 6.15 **Stack usage: (a) before and (b) after call**

registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*. This section describes ARM machine language and the tedious process of converting between assembly and machine language.

ARM uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would also encourage a single instruction format, but that is too restrictive. However, this issue allows us to introduce the last design principle:

> **Design Principle 4***:* Good design demands good compromises.

ARM makes the compromise of defining three main instruction formats: *Data-processing*, *Memory*, and *Branch*. This small number of formats allows for some regularity among instructions, and thus simpler decoder hardware, while also accommodating different instruction needs. Data-processing instructions have a first source register, a second source that is either an immediate or a register, possibly shifted, and a destination register. The Data-processing format has several variations for these second sources. Memory instructions have three operands: a base register, an offset that is either an immediate or an optionally shifted register, and a register that is the destination on an `LDR` and another source on an `STR`. Branch instructions take one 24-bit immediate branch offset. This section discusses these ARM instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all the ARMv4 instructions.

6.4.1 Data-processing Instructions

The data-processing instruction format is the most common. The first source operand is a register. The second source operand can be an immediate or an optionally shifted register. A third register is the destination. Figure 6.16 shows the data-processing instruction format. The 32-bit instruction has six fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*.

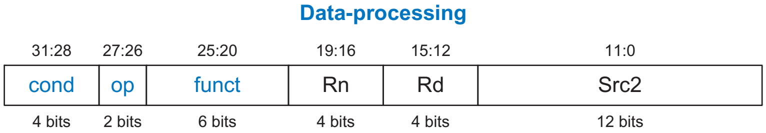

Data-processing
31:28
27:26
25:20
19:16
15:12
11:0
cond
op
funct
Rn
Rd
Src2
4 bits
2 bits
6 bits
4 bits
4 bits
12 bits

Figure 6.16 **Data-processing instruction format**

The operation the instruction performs is encoded in the fields highlighted in blue: *op* (also called the opcode or operation code) and *funct* or function code; the *cond* field encodes conditional execution based on flags described in Section 6.3.2. Recall that $cond = 1110_2$ for unconditional instructions. *op* is 00_2 for data-processing instructions.

The operands are encoded in the three fields: *Rn*, *Rd*, and *Src2*. *Rn* is the first source register and *Src2* is the second source; *Rd* is the destination register.

Figure 6.17 shows the format of the *funct* field and the three variations of *Src2* for data-processing instructions. *funct* has three subfields: *I*, *cmd*, and *S*. The *I*-bit is 1 when *Src2* is an immediate. The *S*-bit is 1 when the instruction sets the condition flags. For example, `SUBS R1, R9, #11` has $S = 1$. *cmd* indicates the specific data-processing instruction, as given in Table B.1 in Appendix B. For example, *cmd* is 4 (0100_2) for `ADD` and 2 (0010_2) for `SUB`.

Three variations of *Src2* encoding allow the second source operand to be (1) an immediate, (2) a register (*Rm*) optionally shifted by a constant (*shamt5*), or (3) a register (*Rm*) shifted by another register (*Rs*). For the latter two encodings of *Src2*, *sh* encodes the type of shift to perform, as will be shown in Table 6.8.

Rd is short for "register destination." *Rn* and *Rm* unintuitively indicate the first and second register sources.

Data-processing instructions have an unusual immediate representation involving an 8-bit unsigned immediate, *imm8*, and a 4-bit rotation, *rot*. *imm8* is rotated right by $2 \times rot$ to create a 32-bit constant. Table 6.7 gives example rotations and resulting 32-bit constants for the 8-bit immediate 0xFF. This representation is valuable because it

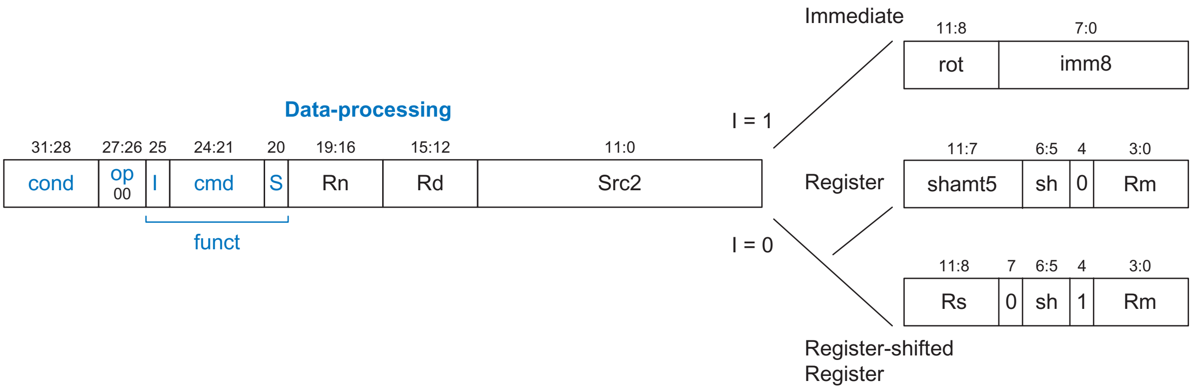

Data-processing
31:28
27:26
25
24:21
20
19:16
15:12
11:0
cond
op
00
I
cmd
S
Rn
Rd
Src2
funct
I = 1
I = 0
Immediate
11:8
7:0
rot
imm8
Register
11:7
6:5
4
3:0
shamt5
sh
0
Rm
11:8
7
6:5
4
3:0
Rs
0
sh
1
Rm
Register-shifted Register

Figure 6.17 **Data-processing instruction format showing the *funct* field and *Src2* variations**

Table 6.7 Immediate rotations and resulting 32-bit constant for *imm8* = 0xFF

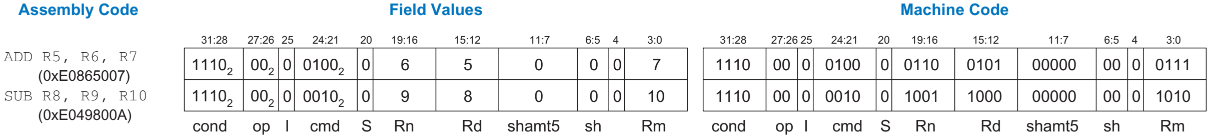
rot	32-bit Constant
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
…	…
1111	0000 0000 0000 0000 0000 0011 1111 1100

If an immediate has multiple possible encodings, the representation with the smallest rotation value *rot* is used. For example, #12 would be represented as (*rot*, *imm8*) = (0000, 00001100), not (0001, 00110000).

permits many useful constants, including small multiples of any power of two, to be packed into a small number of bits. Section 6.6.1 describes how to generate arbitrary 32-bit constants.

Figure 6.18 shows the machine code for `ADD` and `SUB` when *Src2* is a register. The easiest way to translate from assembly to machine code is to write out the values of each field and then convert these values to binary. Group the bits into blocks of four to convert to hexadecimal to make the machine language representation more compact. Beware that the destination is the first register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. *Rn* and *Rm* are the first and second source operands, respectively. For example, the assembly instruction `ADD R5, R6, R7` has *Rn* = 6, *Rd* = 5, and *Rm* = 7.

Figure 6.19 shows the machine code for `ADD` and `SUB` with an immediate and two register operands. Again, the destination is the first



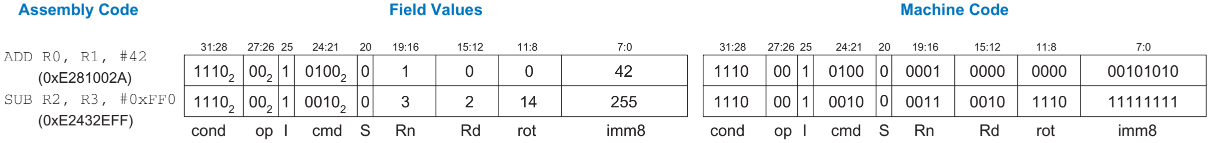
Assembly Code / **Field Values**

Assembly Code	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
`ADD R5, R6, R7` (0xE0865007)	1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
`SUB R8, R9, R10` (0xE049800A)	1110_2	00_2	0	0010_2	0	9	8	0	0	0	10
	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0100	0	0110	0101	00000	00	0	0111
1110	00	0	0010	0	1001	1000	00000	00	0	1010
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Figure 6.18 Data-processing instructions with three register operands



Assembly Code / **Field Values**

Assembly Code	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
`ADD R0, R1, #42` (0xE281002A)	1110_2	00_2	1	0100_2	0	1	0	0	42
`SUB R2, R3, #0xFF0` (0xE2432EFF)	1110_2	00_2	1	0010_2	0	3	2	14	255
	cond	op	I	cmd	S	Rn	Rd	rot	imm8

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110	00	1	0100	0	0001	0000	0000	00101010
1110	00	1	0010	0	0011	0010	1110	11111111
cond	op	I	cmd	S	Rn	Rd	rot	imm8

Figure 6.19 Data-processing instructions with an immediate and two register operands

register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. The immediate of the `ADD` instruction (42) can be encoded in 8 bits, so no rotation is needed (*imm8* = 42, *rot* = 0). However, the immediate of `SUB R2, R3, 0xFF0` cannot be encoded directly using the 8 bits of *imm8*. Instead, *imm8* is 255 (0xFF), and it is rotated right by 28 bits (*rot* = 14). This is easiest to interpret by remembering that the right rotation by 28 bits is equivalent to a left rotation by 32−28 = 4 bits.

Shifts are also data-processing instructions. Recall from Section 6.3.1 that the amount by which to shift can be encoded using either a 5-bit immediate or a register.

Figure 6.20 shows the machine code for logical shift left (`LSL`) and rotate right (`ROR`) with immediate shift amounts. The *cmd* field is 13 (1101_2) for all shift instruction, and the shift field (*sh*) encodes the type of shift to perform, as given in Table 6.8. *Rm* (i.e., R5) holds the 32-bit value to be shifted, and *shamt5* gives the number of bits to shift. The shifted result is placed in *Rd*. *Rn* is not used and should be 0.

Figure 6.21 shows the machine code for `LSR` and `ASR` with the shift amount encoded in the least significant 8 bits of *Rs* (R6 and R12). As

Table 6.8 ***sh*** **field encodings**

Instruction	sh	Operation
`LSL`	00_2	Logical shift left
`LSR`	01_2	Logical shift right
`ASR`	10_2	Arithmetic shift right
`ROR`	11_2	Rotate right

Assembly Code / **Field Values**

Assembly Code	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
`LSL R0, R9, #7` (0xE1A00389)	1110_2	00_2	0	1101_2	0	0	0	7	00_2	0	9
`ROR R3, R5, #21` (0xE1A03AE5)	1110_2	00_2	0	1101_2	0	0	3	21	11_2	0	5
	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	1101	0	0000	0000	00111	00	0	1001
1110	00	0	1101	0	0000	0011	10101	11	0	0101
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Figure 6.20 **Shift instructions with immediate shift amounts**

Assembly Code / **Field Values**

Assembly Code	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
`LSR R4, R8, R6` (0xE1A04638)	1110_2	00_2	0	1101_2	0	0	4	6	0	01_2	1	8
`ASR R5, R1, R12` (0xE1A05C51)	1110_2	00_2	0	1101_2	0	0	5	12	0	10_2	1	1
	cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
1110	00	0	1101	0	0000	0100	0110	0	01	1	1000
1110	00	0	1101	0	0000	0101	1100	0	10	1	0001
cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

Figure 6.21 **Shift instructions with register shift amounts**

Table 6.9 **Offset type control bits for memory instructions**

	Meaning	
Bit	$\overline{I}$	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

before, *cmd* is 13 (1101_2), *sh* encodes the type of shift, *Rm* holds the value to be shifted, and the shifted result is placed in *Rd*. This instruction uses the *register-shifted register* addressing mode, where one register (*Rm*) is shifted by the amount held in a second register (*Rs*). Because the least significant 8 bits of *Rs* are used, *Rm* can be shifted by up to 255 positions. For example, if *Rs* holds the value 0xF001001C, the shift amount is 0x1C (28). A logical shift by more than 31 bits pushes all the bits off the end and produces all 0's. Rotate is cyclical, so a rotate by 50 bits is equivalent to a rotate by 18 bits.

6.4.2 Memory Instructions

Memory instructions use a format similar to that of data-processing instructions, with the same six overall fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*, as shown in Figure 6.22. However, memory instructions use a different *funct* field encoding, have two variations of *Src2*, and use an *op* of 01_2. *Rn* is the base register, *Src2* holds the offset, and *Rd* is the destination register in a load or the source register in a store. The offset is either a 12-bit unsigned immediate (*imm12*) or a register (*Rm*) that is optionally shifted by a constant (*shamt5*). *funct* is composed of six control bits: $\overline{I}$, *P*, *U*, *B*, *W*, and *L*. The $\overline{I}$ (immediate) and *U* (add) bits determine whether the offset is an immediate or register and whether it should be added or subtracted, according to Table 6.9. The *P* (pre-index) and *W* (writeback) bits specify the index mode according to Table 6.10. The *L* (load) and *B* (byte) bits specify the type of memory operation according to Table 6.11.

Table 6.10 **Index mode control bits for memory instructions**

P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

Table 6.11 **Memory operation type control bits for memory instructions**

L	B	Instruction
0	0	`STR`
0	1	`STRB`
1	0	`LDR`
1	1	`LDRB`

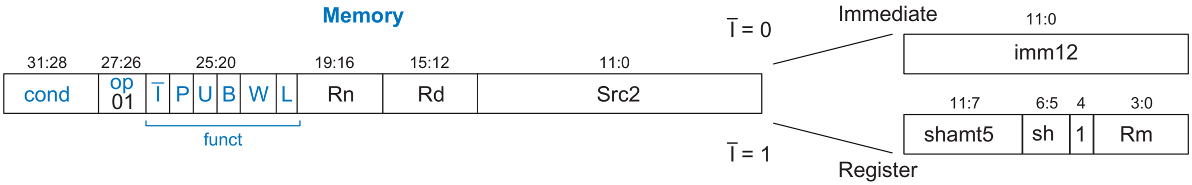


Figure 6.22 **Memory instruction format for** `LDR`, `STR`, `LDRB`, **and** `STRB`

Example 6.3 TRANSLATING MEMORY INSTRUCTIONS INTO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

`STR R11, [R5], #-26`

Notice the counterintuitive encoding of post-indexing mode.

Solution: `STR` is a memory instruction, so it has an *op* of 01_2. According to Table 6.11, $L = 0$ and $B = 0$ for `STR`. The instruction uses post-indexing, so according to Table 6.10, $P = 0$ and $W = 0$. The immediate offset is subtracted from the base, so $\bar{I} = 0$ and $U = 0$. Figure 6.23 shows each field and the machine code. Hence, the machine language instruction is 0xE405B01A.

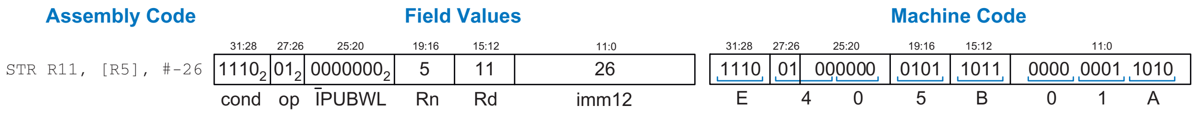

Assembly Code
Field Values
Machine Code
STR R11, [R5], #-26
31:28
27:26
25:20
19:16
15:12
11:0
1110₂
01₂
000000₂
5
11
26
cond
op
ĪPUBWL
Rn
Rd
imm12
1110
01
000000
0101
1011
0000
0001
1010
E
4
0
5
B
0
1
A

Figure 6.23 Machine code for the memory instruction of Example 6.3

6.4.3 Branch Instructions

Branch instructions use a single 24-bit signed immediate operand, *imm24*, as shown in Figure 6.24. As with data-processing and memory instructions, branch instructions begin with a 4-bit condition field and a 2-bit *op*, which is 10_2. The *funct* field is only 2 bits. The upper bit of *funct* is always 1 for branches. The lower bit, *L*, indicates the type of branch operation: 1 for `BL` and 0 for `B`. The remaining 24-bit two's complement *imm24* field is used to specify an instruction address relative to PC + 8.

Code Example 6.28 shows the use of the branch if less than (`BLT`) instruction and Figure 6.25 shows the machine code for that instruction. The *branch target address* (*BTA*) is the address of the next instruction to execute if the branch is taken. The `BLT` instruction in Figure 6.25 has a BTA of 0x80B4, the instruction address of the `THERE` label.

The 24-bit immediate field gives the number of instructions between the BTA and PC + 8 (two instructions past the branch). In this case, the value in the immediate field (*imm24*) of `BLT` is 3 because the BTA (0x80B4) is three instructions past PC + 8 (0x80A8).

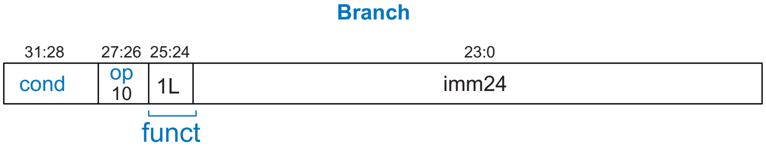

Branch
31:28
27:26
25:24
23:0
cond
op
10
1L
imm24
funct

Figure 6.24 Branch instruction format

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

ARM Assembly Code

```
0x80A0         BLT THERE
0x80A4         ADD R0, R1, R2
0x80A8         SUB R0, R0, R9
0x80AC         ADD SP, SP, #8
0x80B0         MOV PC, LR
0x80B4 THERE   SUB R0, R0, #1
0x80B8         ADD R3, R3, #0x5
```

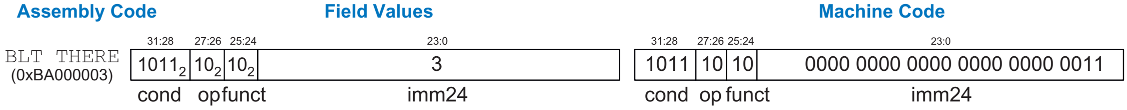

Assembly Code
Field Values
Machine Code
BLT THERE
(0xBA000003)
31:28
27:26
25:24
23:0
1011_2
10_2
10_2
3
cond
op
funct
imm24
31:28
27:26
25:24
23:0
1011
10
10
0000 0000 0000 0000 0000 0011
cond
op
funct
imm24

Figure 6.25 Machine code for branch if less than (BLT)

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC + 8.

Example 6.4 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

```
0x8040 TEST   LDRB  R5, [R0, R3]
0x8044        STRB  R5, [R1, R3]
0x8048        ADD   R3, R3, #1
0x8044        MOV   PC, LR
0x8050        BL    TEST
0x8054        LDR   R3, [R1], #4
0x8058        SUB   R4, R3, #9
```

Solution: Figure 6.26 shows the machine code for the branch and link instruction (BL). Its branch target address (0x8040) is six instructions behind PC + 8 (0x8058), so the immediate field is -6.

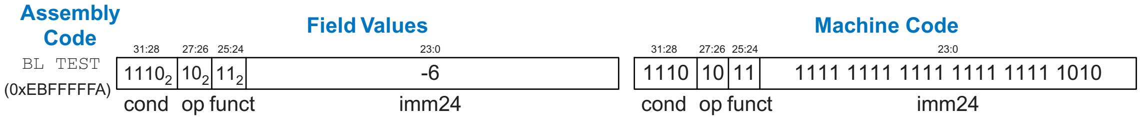

Assembly Code
Field Values
Machine Code
BL TEST
(0xEBFFFFFA)
31:28
27:26
25:24
23:0
1110_2
10_2
11_2
-6
cond
op
funct
imm24
31:28
27:26
25:24
23:0
1110
10
11
1111 1111 1111 1111 1111 1010
cond
op
funct
imm24

Figure 6.26 BL machine code

ARM is unusual among RISC architectures in that it allows the second source operand to be shifted in register and base addressing modes. This requires a shifter in series with the ALU in the hardware implementation but significantly reduces code length in common programs, especially array accesses. For example, in an array of 32-bit data elements, the array index must be left-shifted by 2 to compute the byte offset into the array. Any type of shift is permitted, but left shifts for multiplication are most common.

6.4.4 Addressing Modes

This section summarizes the modes used for addressing instruction operands. ARM uses four main modes: register, immediate, base, and PC-relative addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you easily learn other assembly languages. Register and base addressing have several submodes described below. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC). Table 6.12 summarizes and gives examples of each addressing mode.

Data-processing instructions use register or immediate addressing, in which the first source operand is a register and the second is a register or immediate, respectively. ARM allows the second register to be optionally shifted by an amount specified in an immediate or a third register. Memory instructions use base addressing, in which the base address comes from a register and the offset comes from an immediate, a register, or a register shifted by an immediate. Branches use PC-relative addressing in which the branch target address is computed by adding an offset to PC + 8.

6.4.5 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all

Table 6.12 **ARM operand addressing modes**

Operand Addressing Mode	Example	Description
Register		
Register-only	`ADD R3, R2, R1`	`R3 ← R2 + R1`
Immediate-shifted register	`SUB R4, R5, R9, LSR #2`	`R4 ← R5 − (R9 >> 2)`
Register-shifted register	`ORR R0, R10, R2, ROR R7`	`R0 ← R10 | (R2 ROR R7)`
Immediate	`SUB R3, R2, #25`	`R3 ← R2 − 25`
Base		
Immediate offset	`STR R6, [R11, #77]`	`mem[R11+77] ← R6`
Register offset	`LDR R12, [R1, −R5]`	`R12 ← mem[R1 − R5]`
Immediate-shifted register offset	`LDR R8, [R9, R2, LSL #2]`	`R8 ← mem[R9 + (R2 << 2)]`
PC-Relative	`B LABEL1`	Branch to `LABEL1`

formats start with a 4-bit condition field and a 2-bit *op*. The best place to begin is to look at the *op*. If it is 00_2, then the instruction is a data-processing instruction; if it is 01_2, then the instruction is a memory instruction; if it is 10_2, then it is a branch instruction. Based on that, the rest of the fields can be interpreted.

Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0xE0475001
0xE5949010
```

Solution: First, we represent each instruction in binary and look at bits 27:26 to find the *op* for each instruction, as shown in Figure 6.27. The *op* fields are 00_2 and 01_2, indicating a data-processing and memory instruction, respectively. Next, we look at the *funct* field of each instruction.

The *cmd* field of the data-processing instruction is 2 (0010_2) and the *I*-bit (bit 25) is 0, indicating that it is a `SUB` instruction with a register *Src2*. *Rd* is 5, *Rn* is 7, and *Rm* is 1.

The *funct* field for the memory instruction is 011001_2. $B = 0$ and $L = 1$, so this is an `LDR` instruction. $P = 1$ and $W = 0$, indicating offset addressing. $\bar{I} = 0$, so the offset is an immediate. $U = 1$, so the offset is added. Thus, it is a load register instruction with an immediate offset that is added to the base register. *Rd* is 9, *Rn* is 4, and *imm12* is 16. Figure 6.27 shows the assembly code equivalent of the two machine instructions.

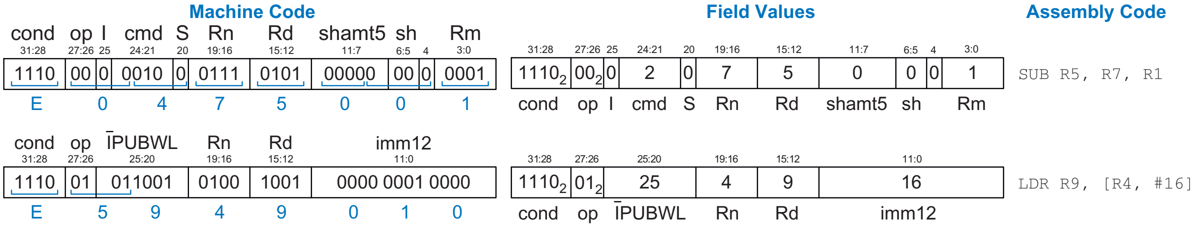
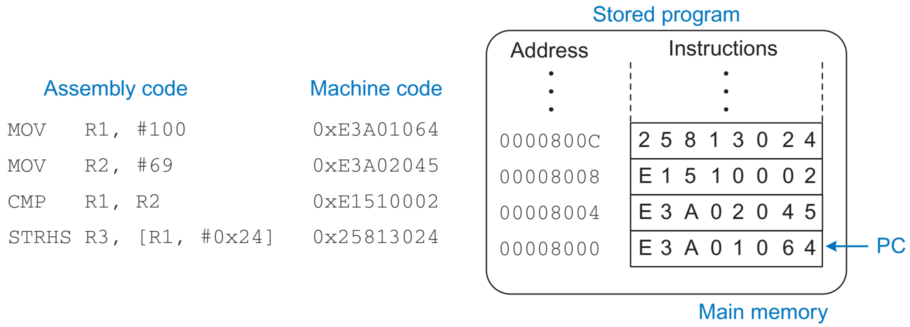

Machine Code
cond op I cmd S Rn Rd shamt5 sh Rm
31:28 27:26 25 24:21 20 19:16 15:12 11:7 6:5 4 3:0
1110 00 0 0010 0 0111 0101 00000 00 0 0001
E 0 4 7 5 0 0 1
cond op ĪPUBWL Rn Rd imm12
31:28 27:26 25:20 19:16 15:12 11:0
1110 01 011001 0100 1001 0000 0001 0000
E 5 9 4 9 0 1 0
Field Values
1110₂ 00₂ 0 2 0 7 5 0 0 0 1
cond op I cmd S Rn Rd shamt5 sh Rm
1110₂ 01₂ 25 4 9 16
cond op ĪPUBWL Rn Rd imm12
Assembly Code
SUB R5, R7, R1
LDR R9, [R4, #16]

Figure 6.27 **Machine code to assembly code translation**

6.4.6 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different



Assembly code

```
MOV   R1, #100            
MOV   R2, #69             
CMP   R1, R2              
STRHS R3, [R1, #0x24]     
```

Machine code

```
0xE3A01064
0xE3A02045
0xE1510002
0x25813024
```

Figure 6.28 **Stored program**

program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory reads and instruction executions.

Figure 6.28 shows how machine instructions are stored in memory. In ARM programs, the instructions are normally stored starting at low addresses, in this case 0x00008000. Remember that ARM memory is byte-addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC), which is register R15. For historical reasons, a read to the PC returns the address of the current instruction plus 8.

To execute the code in Figure 6.28, the PC is initialized to address 0x00008000. The processor fetches the instruction at that memory address and executes the instruction, 0xE3A01064 (`MOV R1, #100`). The processor then increments the PC by 4 to 0x00008004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For ARM, the architectural state includes the register file and status registers. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

Ada Lovelace, 1815–1852. A British mathematician who wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the daughter of the poet Lord Byron.

6.5 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING*

Until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution. We begin by introducing an example ARM *memory map*, which defines where code, data, and stack memory are located.

Figure 6.29 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

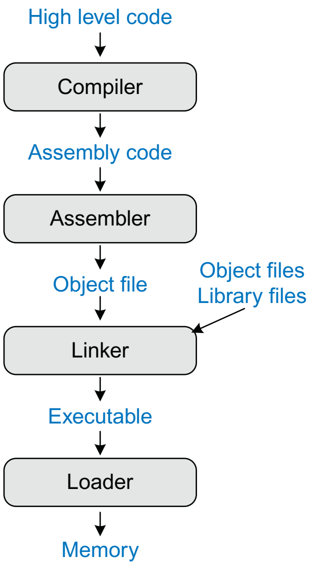

High level code
Compiler
Assembly code
Assembler
Object file
Object files
Library files
Linker
Executable
Loader
Memory

Figure 6.29 **Steps for translating and starting a program**

6.5.1 The Memory Map

With 32-bit addresses, the ARM address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFFC. Figure 6.30 shows an example memory map. The ARM architecture divides the address space into five parts or segments: the text segment,

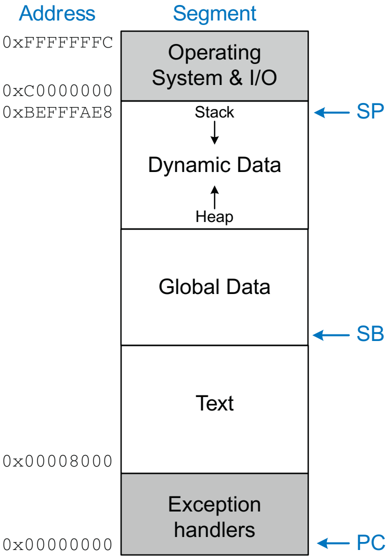

Address
Segment
0xFFFFFFFC
Operating System & I/O
0xC0000000
0xBEFFFAE8
Stack
SP
Dynamic Data
Heap
Global Data
SB
Text
0x00008000
Exception handlers
0x00000000
PC

Figure 6.30 **Example ARM memory map**

We present an example ARM memory map here; however, in ARM, the memory map is somewhat flexible. While the exception vector table must be located at 0x0 and memory-mapped I/O is typically located at the high memory addresses, the user can define where the text (code and constant data), stack, and global data are placed. Moreover, at least historically, most ARM systems have less than 4 GB of memory.

global data segment, dynamic data segment, and segments for exception handlers, the operating system (OS) and input/output (I/O). The following sections describe each segment.

The Text Segment

The *text segment* stores the machine language program. ARM also calls this the *read-only* (RO) *segment*. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Global variables are allocated in memory before the program begins executing. ARM also calls this the *read/write* (RW) *segment*. Global variables are typically accessed using a *static base* register that points to the start of the global segment. ARM conventionally uses R9 as the static base pointer (SB).

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the heap. The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program.

Upon start-up, the operating system sets up the stack pointer (SP) to point to the top of the stack. The stack typically grows downward, as shown here. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers. As discussed in Section 6.3.7, functions also use the stack to save and restore registers. Each stack frame is accessed in last-in-first-out order.

The *heap* stores data that is allocated by the program during runtime. In C, memory allocations are made by the `malloc` function; in C++ and Java, `new` is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap typically grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

The Exception Handler, OS, and I/O Segments

The lowest part of the ARM memory map is reserved for the exception vector table and exception handlers, starting at address 0x0 (see Section 6.6.3). The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see Section 9.2).

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal.

6.5.2 Compilation

A compiler translates high-level code into assembly language. The examples in this section are based on GCC, a popular and widely used free compiler, running on the Raspberry Pi single-board computer

Code Example 6.29 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables


int sum(int a, int b) {
  return (a + b);
}


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
```

ARM Assembly Code

```
  .text
  .global  sum
  .type    sum, %function
sum:
  add      r0, r0, r1
  bx       lr
  .global  main
  .type    main, %function
main:
  push     {r3, lr}
  mov      r0, #2
  ldr      r3, .L3
  str      r0, [r3, #0]
  mov      r1, #3
  ldr      r3, .L3+4
  str      r1, [r3, #0]
  bl       sum
  ldr      r3, .L3+8
  str      r0, [r3, #0]
  pop      {r3, pc}
.L3:
  .word    f
  .word    g
  .word    y
```

In Code Example 6.29, global variables are accessed using two memory instructions: one to load the *address* of the variable, and a second to read or write the variable. The addresses of the global variables are placed after the code, starting at label `.L3`. `LDR R3, .L3` loads the *address* of `f` into `R3`, and `STR R0, [R3, #0]` writes to `f`; `LDR R3, .L3+4` loads the *address* of `g` into `R3`, and `STR R1, [R3, #0]` writes to `g`, and so on. Section 6.6.1 describes this assembly code construct further.

(see Section 9.3). Code Example 6.29 shows a simple high-level program with three global variables and two functions, along with the assembly code produced by GCC.

To compile, assemble, and link a C program named `prog.c` with GCC, use the command:

```
gcc -O1 -g prog.c -o prog
```

This command produces an executable output file called `prog`. The `-O1` flag asks the compiler to perform basic optimizations rather than producing grossly inefficient code. The `-g` flag tells the compiler to include debugging information in the file.

To see the intermediate steps, we can use GCC's `-S` flag to compile but not assemble or link.

```
gcc -O1 -S prog.c -o prog.s
```

The output, `prog.s`, is rather verbose, but the interesting parts are shown in Code Example 6.29. Note that GCC requires labels to be followed by a colon. The GCC output is in lowercase and has other assembler directives not discussed here. Observe that `sum` returns using the `BX` instruction rather than `MOV PC, LR`. Also, observe that GCC elected to save and restore R3 even though it is not one of the preserved registers. The addresses of the global variables will be stored in a table starting at label .L3.

6.5.3 Assembling

An assembler turns the assembly language code into an *object file* containing machine language code. GCC can create the object file from either `prog.s` or directly from `prog.c` using

```
gcc -c prog.s -o prog.o
```

or

```
gcc -O1 -g -c prog.c -o prog.o
```

The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names. The names and addresses of the symbols are kept in a symbol table. On the second pass through the code, the assembler produces the machine language code. Addresses for labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

We can *disassemble* the object file using the `objdump` command to see the assembly language code beside the machine language code. If the code was originally compiled with `–g`, the disassembler also shows the corresponding lines of C code:

```
objdump -S prog.o
```

The following shows the disassembly of section .text:

```
00000000 <sum>:

int sum(int a, int b) {
   return (a + b);
}
  0:  e0800001   add  r0, r0, r1
  4:  e12fff1e   bx  lr

00000008 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8:   e92d4008   push  {r3, lr}
       f = 2;
  c:   e3a00002   mov   r0, #2
  10:  e59f301c   ldr   r3, [pc, #28]  ; 34 <main+0x2c>
  14:  e5830000   str   r0, [r3]
       g = 3;
  18:  e3a01003   mov   r1, #3
  1c:  e59f3014   ldr   r3, [pc, #20]  ; 38 <main+0x30>
  20:  e5831000   str   r1, [r3]
       y = sum(f,g);
  24:  ebfffffe   bl    0 <sum>
```

Recall from Section 6.4.6 that a read to PC returns the address of the current instruction plus 8. So, `LDR R3, [PC, #28]` loads f's address, which is just after the code at: (PC + 8) + 28 = (0x10 + 0x8) + 0x1C = 0x34.

```
  28:  e59f300c   ldr   r3, [pc, #12]  ; 3c <main+0x34>
  2c:  e5830000   str   r0, [r3]
  return y;
}
  30:  e8bd8008   pop  {r3, pc}
  ...
```

We can also view the symbol table from the object file using `objdump` with the `-t` flag. The interesting parts are shown below. Observe that the `sum` function starts at address 0 and has a size of 8 bytes. `main` starts at address 8 and has size 0x38. The global variable symbols `f`, `g`, and `h` are listed and are 4 bytes each, but they have not yet been assigned addresses.

```
objdump -t prog.o
```

```
SYMBOL TABLE:
00000000 l d .text  00000000 .text
00000000 l d .data  00000000 .data
00000000 g F .text  00000008 sum
00000008 g F .text  00000038 main
00000004   O *COM*  00000004 f
00000004   O *COM*  00000004 g
00000004   O *COM*  00000004 y
```

6.5.4 Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated. Also, a program typically involves some start-up code to initialize the stack, heap, and so forth, that must be executed before calling the `main` function.

The job of the linker is to combine all of the object files and the start-up code into one machine language file called the *executable* and assign addresses for global variables. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the code based on the new label and global variable addresses. Invoke GCC to link the object file using:

```
gcc prog.o -o prog
```

We can again disassemble the executable using:

```
objdump -S -t prog
```

The start-up code is too lengthy to show, but our program begins at address 0x8390 in the text segment and the global variables are assigned addresses starting at 0x10570 in the global segment. Notice the `.word` assembler directives defining the addresses of the global variables `f`, `g`, and `y`.

```
00008390 <sum>:

int sum(int a, int b) {
  return (a + b);
}
  8390:  e0800001   add  r0, r0, r1
  8394:  e12fff1e   bx  lr

00008398 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8398:  e92d4008   push  {r3, lr}
  f = 2;
  839c:  e3a00002   mov  r0, #2
  83a0:  e59f301c   ldr  r3, [pc, #28]  ; 83c4 <main+0x2c>
  83a4:  e5830000   str  r0, [r3]
  g = 3;
  83a8:  e3a01003   mov  r1, #3
  83ac:  e59f3014   ldr  r3, [pc, #20]  ; 83c8 <main+0x30>
  83b0:  e5831000   str  r1, [r3]
  y = sum(f,g);
  83b4:  ebfffff5   bl  8390 <sum>
  83b8:  e59f300c   ldr  r3, [pc, #12]  ; 83cc <main+0x34>
  83bc:  e5830000   str  r0, [r3]
  return y;
}
  83c0:  e8bd8008   pop   {r3, pc}
  83c4:  00010570   .word  0x00010570
  83c8:  00010574   .word  0x00010574
  83cc:  00010578   .word  0x00010578
```

The instruction `LDR R3, [PC, #28]` in the executable loads from address (PC + 8) + 28 = (0x83A0 + 0x8) + 0x1C = 0x83C4. This memory address contains the value 0x10570, the location of global variable `f`.

The executable also contains an updated symbol table with the relocated addresses of the functions and global variables.

```
SYMBOL TABLE:
000082e4 l   d  .text  00000000   .text
00010564 l   d  .data  00000000   .data
00008390 g    F .text  00000008   sum
00008398 g    F .text  00000038   main
00010570 g    O .bss   00000004   f
00010574 g    O .bss   00000004   g
00010578 g    O .bss   00000004   y
```

6.5.5 Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system jumps to the beginning of the program to begin executing. Figure 6.31 shows the memory map at the beginning of program execution.

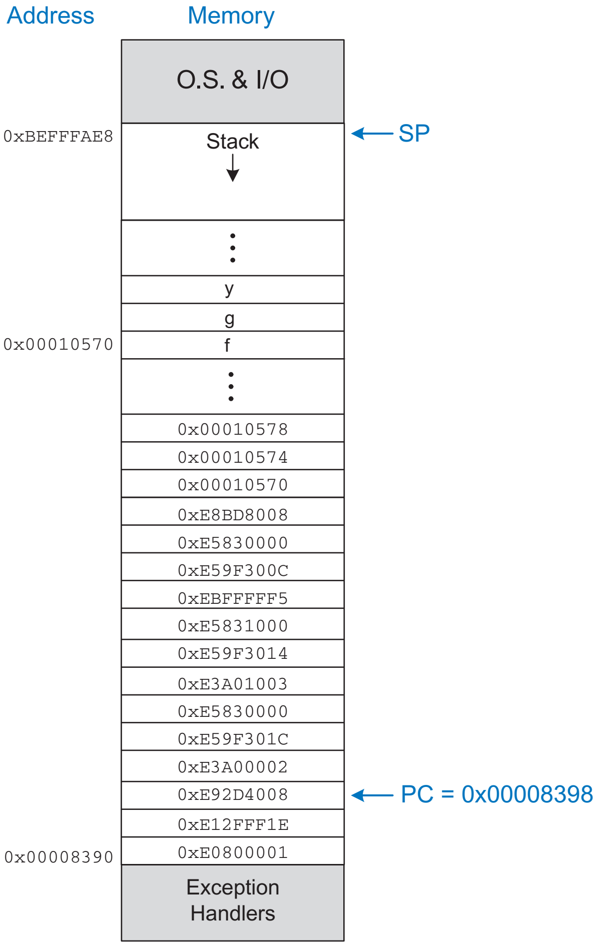

Address
Memory
O.S. & I/O
0xBEFFFAE8
Stack
SP
y
g
0x00010570
f
0x00010578
0x00010574
0x00010570
0xE8BD8008
0xE5830000
0xE59F300C
0xEBFFFFF5
0xE5831000
0xE59F3014
0xE3A01003
0xE5830000
0xE59F301C
0xE3A00002
0xE92D4008
PC = 0x00008398
0xE12FFF1E
0x00008390
0xE0800001
Exception Handlers

Figure 6.31 **Executable loaded in memory**

6.6 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include loading 32-bit literals, NOPs, and exceptions.

6.6.1 Loading Literals

Many programs need to load 32-bit literals, such as constants or addresses. `MOV` only accepts a 12-bit source, so the `LDR` instruction is used to load these numbers from a *literal pool* in the text segment. ARM assemblers accept loads of the form

```
LDR Rd, =literal
LDR Rd, =label
```

The first loads a 32-bit constant specified by `literal`, and the second loads the address of a variable or pointer in the program specified by `label`. In both cases, the value to load is kept in a *literal pool*, which is a portion of the text segment containing literals. The literal pool must be less than 4096 bytes from the `LDR` instruction so that the load can be performed as `LDR Rd, [PC, #offset_to_literal]`. The program must be careful to branch around the literal pool because executing literals would be nonsensical or worse.

Code Example 6.30 illustrates loading a literal. As shown in Figure 6.32, suppose the `LDR` instruction is at address 0x8110 and the literal is at 0x815C. Remember that reading the PC returns the address 8 bytes beyond the current instruction being executed. Hence, when the `LDR` is executed, reading the PC returns 0x8118. Thus, the `LDR` uses an offset of 0x44 to find the literal pool: `LDR R1, [PC, #0x44]`.

Code Example 6.30 LARGE IMMEDIATE USING A LITERAL POOL

High-level code

```
int a = 0x2B9056F;
```

ARM Assembly Code

```
; R1 = a
  LDR R1, =0x2B9056F
  ...
```

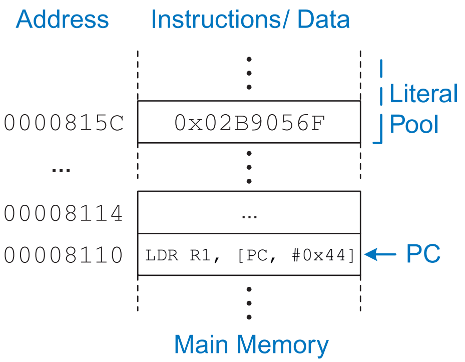


Figure 6.32 Example literal pool

Pseudoinstructions are not actually part of the instruction set but are shorthand for instructions or instruction sequences that are commonly used by programmers and compilers. The assembler translates pseudoinstructions into one or more actual instructions.

6.6.2 NOP

NOP is a mnemonic for "no operation" and is pronounced "no op." It is a *pseudoinstruction* that does nothing. The assembler translates it to `MOV R0, R0` (0xE1A00000). NOPs are useful to, among other things, achieve some delay or align instructions.

6.6.3 Exceptions

An *exception* is like an unscheduled function call that branches to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, and then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then branches to code in the operating system (OS), which may choose to either emulate the unimplemented instruction or terminate the offending program. Software exceptions are sometimes called *traps*. A particularly important form of a trap is a *system call*, whereby the program invokes a function in the OS running at a higher privilege level. Other causes of exceptions include reset and attempts to read nonexistent memory.

Like any other function call, an exception must save the return address, jump to some address, do its work, clean up after itself, and return to the program where it left off. Exceptions use a *vector* table to determine where to jump to the *exception handler* and use *banked registers* to maintain extra copies of key registers so that they will not corrupt the registers in the active program. Exceptions also change the *privilege level* of the program, allowing the exception handler to access protected parts of memory.

Execution Modes and Privilege Levels

An ARM processor can operate in one of several execution modes with different privilege levels. The different modes allow an exception to take place in an exception handler without corrupting state; for example, an interrupt could occur while the processor is executing operating system code in Supervisor mode, and a subsequent Abort exception could occur if the interrupt attempted to access an invalid memory address. The exception handlers would eventually return and resume the supervisor code. The mode is specified in the bottom bits of the Current Program Status Register (CPSR), as was shown in Figure 6.6. Table 6.13 lists execution modes and their encodings. User mode operates at privilege level PL0, which is unable to access protected portions of memory such as the operating system code. The other modes operate at privilege level PL1, which can access all system resources. Privilege levels are important so that buggy or malicious user code cannot corrupt other programs or crash or infect the system.

Exception Vector Table

When an exception occurs, the processor branches to an offset in the *exception vector table*, depending on the cause of the exception. Table 6.14 describes the vector table, which is normally located starting at address 0x00000000 in memory. For example, when an interrupt occurs, the processor branches to address 0x00000018. Similarly, on

Table 6.13 **ARM execution modes**

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001

Table 6.14 **Exception vector table**

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

power-up, the processor branches to address 0x00000000. Each exception vector offset typically contains a branch instruction to an exception handler, code that handles the exception and then either exits or returns to the user code.

ARM also supports a High Vectors mode in which the exception vector table starts at address 0xFFFF0000. For example, the system may boot using a vector table in ROM at address 0x00000000. Once the system starts up, the OS may write an updated vector table in RAM at 0xFFFF0000 and put the system into High Vectors mode.

Banked Registers

Before an exception changes the PC, it must save the return address in the LR so that the exception handler knows where to return. However, it must take care not to disturb the value already in the LR, which the program will need later. Therefore, the processor maintains a bank of different registers to use as LR during each of the execution modes. Similarly, the exception handler must not disturb the status register bits.

Hence, a bank of *saved program status registers* (SPSRs) is used to hold a copy of the CPSR during exceptions.

If an exception takes place while a program is manipulating its stack frame, the frame might be in an unstable state (e.g., data has been written onto the stack but the stack pointer is not yet pointing to the top of stack). Hence, each execution mode also uses its own stack and banked copy of SP pointing to the top of its stack. Memory must be reserved for each execution mode's stack and banked versions of the stack pointers must be initialized at start-up.

The first thing that an exception handler must do is to push all of the registers it might change onto the stack. This takes some time. ARM has a *fast interrupt* execution mode FIQ in which R8–R12 are also banked. Thus, the exception handler can immediately begin without saving these registers.

Exception Handling

Now that we have defined execution modes, exception vectors, and banked registers, we can define what occurs during an exception. Upon detecting an exception, the processor:

1. Stores the CPSR into the banked SPSR
2. Sets the execution mode and privilege level based on the type of exception
3. Sets *interrupt mask* bits in the CPSR so that the exception handler will not be interrupted
4. Stores the return address into the banked LR
5. Branches to the exception vector table based on the type of exception

The processor then executes the instruction in the exception vector table, typically a branch to the exception handler. The handler usually pushes other registers onto its stack, takes care of the exception, and pops the registers back off the stack. The exception handler returns using the `MOVS PC, LR` instruction, a special flavor of `MOV` that performs the following cleanup:

1. Copies the banked SPSR to the CPSR to restore the status register
2. Copies the banked LR (possibly adjusted for certain exceptions) to the PC to return to the program where the exception occurred
3. Restores the execution mode and privilege level

Exception-Related Instructions

Programs operate at a low privilege level, whereas the operating system has a higher privilege level. To transition between levels in a controlled way, the program places arguments in registers and issues a *supervisor call* (`SVC`) instruction, which generates an exception and raises the privilege level. The OS examines the arguments and performs the requested function, and then returns to the program.

The OS and other code operating at PL1 can access the banked registers for the various execution modes using the `MRS` (move to register from special register) and `MSR` (move to special register from register) instructions. For example, at boot time, the OS will use these instructions to initialize the stacks for exception handlers.

Start-up

On start-up, the processor jumps to the reset vector and begins executing *boot loader* code in supervisor mode. The boot loader typically configures the memory system, initializes the stack pointer, and reads the OS from disk; then it begins a much longer boot process in the OS. The OS eventually will load a program, change to unprivileged user mode, and jump to the start of the program.

6.7 EVOLUTION OF ARM ARCHITECTURE

The ARM1 processor was first developed by Acorn Computer in Britain for the BBC Micro computers in 1985 as an upgrade to the 6502 microprocessor used in many personal computers of the era. It was followed within the year by the ARM2, which went into production in the Acorn Archimedes computer. ARM was an acronym for *Acorn RISC Machine*. The product implemented Version 2 of the ARM instruction set (ARMv2). The address bus was only 26 bits, and the upper 6 bits of the 32-bit PC were used to hold status bits. The architecture included almost all of the instructions described in this chapter, including data-processing, most loads and stores, branches, and multiplies.

ARM soon extended the address bus to a full 32 bits, moving the status bits into a dedicated Current Program Status Register (CPSR). ARMv4, introduced in 1993, added halfword loads and stores and provided both signed and unsigned halfword and byte loads. This is the core of the modern ARM instruction set, and is what we have covered in this chapter.

As of ARMv7, the CPSR is called the Application Program Status Register (APSR).

The ARM instruction set has seen many enhancements described in subsequent sections. The highly successful ARM7TDMI processor in 1995 introduced the 16-bit Thumb instruction set in ARMv4T to improve code density. ARMv5TE added digital signal processing (DSP) and optional floating-point instructions. ARMv6 added multimedia instructions and enhanced the Thumb instruction set. ARMv7 improved the floating-point and multimedia instructions, renaming them Advanced SIMD. ARMv8 introduced a completely new 64-bit architecture. Various other system programming instructions have been introduced as the architecture has evolved.

6.7.1 Thumb Instruction Set

Thumb instructions are 16 bits long to achieve higher code density; they are identical to regular ARM instructions but generally have limitations, including that they:

- Access only the bottom eight registers
- Reuse a register as both a source and destination
- Support shorter immediates
- Lack conditional execution
- Always write the status flags

Almost all ARM instructions have Thumb equivalents. Because the instructions are less powerful, more are required to write an equivalent program. However, the instructions are half as long, giving overall Thumb code size of about 65% of the ARM equivalent. The Thumb instruction set is valuable not only to reduce the size and cost of code storage memory, but also to allow for an inexpensive 16-bit bus to instruction memory and to reduce the power consumed by fetching instructions from the memory.

ARM processors have an instruction set state register, ISETSTATE, that includes a T bit to indicate whether the processor is in normal mode ($T = 0$) or Thumb mode ($T = 1$). This mode determines how instructions should be fetched and interpreted. The `BX` and `BLX` branch instructions toggle the T bit to enter or exit Thumb mode.

Thumb instruction encoding is more complex and irregular than ARM instructions to pack as much useful information as possible into 16-bit halfwords. Figure 6.33 shows encodings for common Thumb instructions. The upper bits specify the type of instruction. Data-processing instructions typically specify two registers, one of which is both the first source and the destination. They always write the status flags. Adds, subtracts, and shifts can specify a short immediate. Conditional branches specify a 4-bit condition code and a short offset, whereas unconditional branches allow a longer offset. Note that `BX` takes a 4-bit register identifier so that it can access the link register LR. Special forms of `LDR`, `STR`, `ADD`, and `SUB` are defined to operate relative to the stack pointer SP (to access the stack frame during function calls). Another special form of `LDR` loads relative to the PC (to access a literal pool). Forms of `ADD` and `MOV` can access all 16 registers. `BL` always requires two halfwords to specify a 22-bit destination.

ARM subsequently refined the Thumb instruction set and added a number of 32-bit Thumb-2 instructions to boost performance of common operations and to allow any program to be written in Thumb mode.

The irregular Thumb instruction set encoding and variable-length instructions (1 or 2 halfwords) are characteristic of 16-bit processor architectures that must pack a large amount of information into a short instruction word. The irregularity complicates instruction decoding.

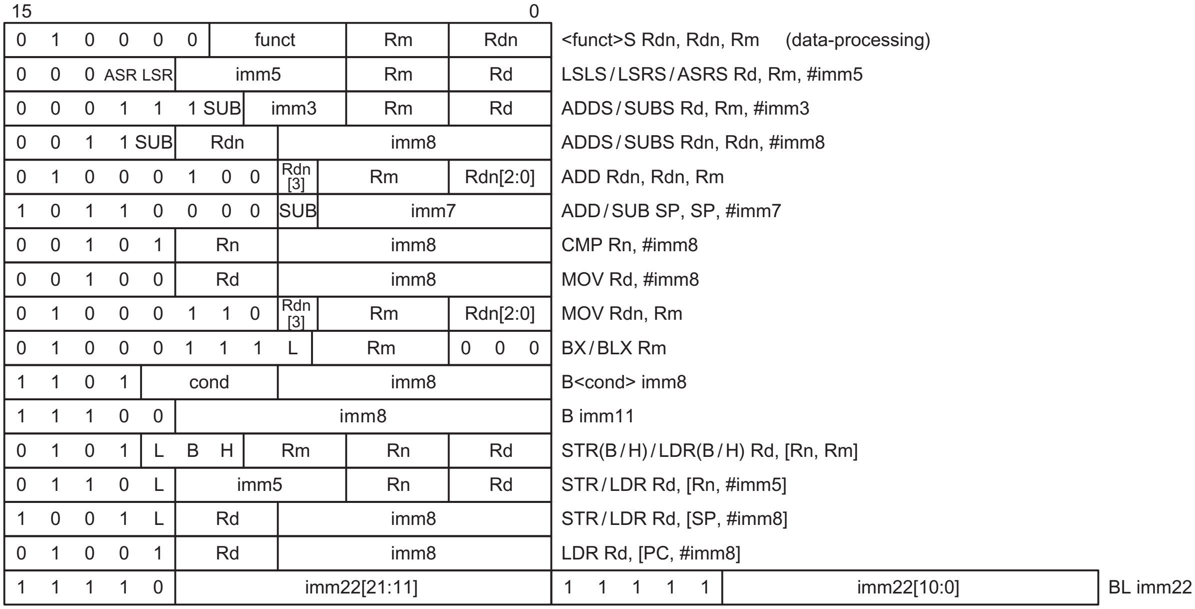

15 0
0 1 0 0 0 0 funct Rm Rdn <funct>S Rdn, Rdn, Rm (data-processing)
0 0 0 ASR LSR imm5 Rm Rd LSLS / LSRS / ASRS Rd, Rm, #imm5
0 0 0 1 1 1 SUB imm3 Rm Rd ADDS / SUBS Rd, Rm, #imm3
0 0 1 1 SUB Rdn imm8 ADDS / SUBS Rdn, Rdn, #imm8
0 1 0 0 0 1 0 0 Rdn [3] Rm Rdn[2:0] ADD Rdn, Rdn, Rm
1 0 1 1 0 0 0 0 SUB imm7 ADD / SUB SP, SP, #imm7
0 0 1 0 1 Rn imm8 CMP Rn, #imm8
0 0 1 0 0 Rd imm8 MOV Rd, #imm8
0 1 0 0 0 1 1 0 Rdn [3] Rm Rdn[2:0] MOV Rdn, Rm
0 1 0 0 0 1 1 1 L Rm 0 0 0 BX / BLX Rm
1 1 0 1 cond imm8 B<cond> imm8
1 1 1 0 0 imm8 B imm11
0 1 0 1 L B H Rm Rn Rd STR(B / H) / LDR(B / H) Rd, [Rn, Rm]
0 1 1 0 L imm5 Rn Rd STR / LDR Rd, [Rn, #imm5]
1 0 0 1 L Rd imm8 STR / LDR Rd, [SP, #imm8]
0 1 0 0 1 Rd imm8 LDR Rd, [PC, #imm8]
1 1 1 1 0 imm22[21:11] 1 1 1 1 1 imm22[10:0] BL imm22

Figure 6.33 Thumb instruction encoding examples

Thumb-2 instructions are identified by their most significant 5 bits being 11101, 11110, or 11111. The processor then fetches a second halfword containing the remainder of the instruction. The Cortex-M series of processors operates exclusively in Thumb state.

6.7.2 DSP Instructions

Digital signal processors (DSPs) are designed to efficiently handle signal processing algorithms such as the Fast Fourier Transform (FFT) and Finite/Infinite Impulse Response filters (FIR/IIR). Common applications include audio and video encoding and decoding, motor control, and speech recognition. ARM provides a number of DSP instructions for these purposes. DSP instructions include multiply, add, and multiply-accumulate (MAC)—multiply and add the result to a running sum: sum = sum + src1 × src2. MAC is a distinguishing feature separating DSP instruction sets from regular instruction sets. It is very commonly used in DSP algorithms and doubles the performance relative to separate multiply and add instructions. However, MAC requires specifying an extra register to hold the running sum.

DSP instructions often operate on short (16-bit) data representing samples read from a sensor by an analog-to-digital converter. However, the intermediate results are held to greater precision (e.g., 32 or 64 bits)

The Fast Fourier Transform (FFT), the most common DSP algorithm, is both complicated and performance-critical. The DSP instructions in computer architectures are intended to perform efficient FFTs, especially on 16-bit fractional data.

The basic multiply instructions, listed in Appendix B, are part of ARMv4. ARMv5TE added the saturating math instructions and packed and fractional multiplies to support DSP algorithms.

Table 6.15 DSP data types

Type	Sign Bit	Integer Bits	Fractional Bits
short	1	15	0
unsigned short	0	16	0
long	1	31	0
unsigned long	0	32	0
long long	1	63	0
unsigned long long	0	64	0
Q15	1	0	15
Q31	1	0	31

or saturated to prevent overflow. In *saturated arithmetic*, results larger than the most positive number are treated as the most positive, and results smaller than the most negative are treated as the most negative. For example, in 32-bit arithmetic, results greater than $2^{31} - 1$ saturate at $2^{31} - 1$, and results less than -2^{31} saturate at -2^{31}. Common DSP data types are given in Table 6.15. Two's complement numbers are indicated as having one sign bit. The 16-, 32-, and 64-bit types are also known as *half*, *single*, and *double* precision, not to be confused with single and double-precision floating-point numbers. For efficiency, two half-precision numbers are packed in a single 32-bit word.

The *integer* types come in signed and unsigned flavors with the sign bit in the msb. *Fractional* types (Q15 and Q31) represent a signed fractional number; for example, Q31 spans the range $[-1, 1-2^{-31}]$ with a step of 2^{-31} between consecutive numbers. These types are not defined in the C standard but are supported by some libraries. Q31 can be converted to Q15 by truncation or rounding. In truncation, the Q15 result is just the upper half. In rounding, 0x00008000 is added to the Q31 value and then the result is truncated. When a computation involves many steps, rounding is useful because it avoids accumulating multiple small truncation errors into a significant error.

ARM added a Q flag to the status registers to indicate that overflow or saturation has occurred in DSP instructions. For applications where accuracy is critical, the program can clear the Q flag before a computation, do the computation in single-precision, and check the Q flag afterward. If it is set, overflow occurred and the computation can be repeated in double precision if necessary.

Saturated arithmetic is an important way to gracefully degrade accuracy in DSP algorithms. Commonly, single-precision arithmetic is sufficient to handle most inputs, but pathological cases can overflow the single-precision range. An overflow causes an abrupt sign change to a radically wrong answer, which may appear to the user as a click in an audio stream or a strangely colored pixel in a video stream. Going to double-precision arithmetic prevents overflow but degrades performance and increases power consumption in the typical case. Saturated arithmetic clips the overflow at the maximum or minimum value, which is usually close to the desired value and causes little inaccuracy.

Addition and subtraction are performed identically no matter which format is used. However, multiplication depends on the type. For example, with 16-bit numbers, the number 0xFFFF is interpreted as 65535 for unsigned short, -1 for short, and -2^{-15} for Q15 numbers. Hence, 0xFFFF × 0xFFFF has a very different value for each representation (4,294,836,225; 1; and 2^{-30}, respectively). This leads to different instructions for signed and unsigned multiplication.

A Q15 number A can be viewed as $a \times 2^{-15}$, where a is its interpretation in the range $[-2^{15}, 2^{15}-1]$ as a signed 16-bit number. Hence, the product of two Q15 numbers is:

$$A \times B = a \times b \times 2^{-30} = 2 \times a \times b \times 2^{-31}$$

This means that to multiply two Q15 numbers and get a Q31 result, do ordinary signed multiplication and then double the product. The product can then be truncated or rounded to put it back into Q15 format if necessary.

The rich assortment of multiply and multiply-accumulate instructions are summarized in Table 6.16. MACs require up to four registers: *RdHi*, *RdLo*, *Rn*, and *Rm*. For double-precision operations, *RdHi* and *RdLo* hold the most and least significant 32 bits, respectively. For example, `UMLAL RdLo, RdHi, Rn, Rm` computes $\{RdHi, RdLo\} = \{RdHi, RdLo\} + Rn \times Rm$. Half-precision multiplies come in various flavors denoted in braces to choose the operands from the top or bottom half of the word, and in *dual* forms where both the top and bottom halves are multiplied. MACs involving half-precision inputs and a single-precision accumulator (`SMLA*`, `SMLAW*`, `SMUAD`, `SMUSD`, `SMLAD`, `SMLSD`) will set the *Q* flag if the accumulator overflows. The most significant word (MSW) multiplies also come in forms with an `R` suffix that round rather than truncate.

The DSP instructions also include saturated add (`QADD`) and subtract (`QSUB`) of 32-bit words that saturate the results instead of overflowing. They also include `QDADD` and `QDSUB`, which double the second operand before adding/subtracting it to/from the first with saturation; we will shortly find these valuable in fractional MACs. They set the *Q* flag if saturation occurs.

Finally, the DSP instructions include `LDRD` and `STRD` that load and store an even/odd pair of registers in a 64-bit memory double word. These instructions increase the efficiency of moving double-precision values between memory and registers.

Table 6.17 summarizes how to use the DSP instructions to multiply or MAC various types of data. The examples assume halfword data is in the bottom half of a register and that the top half is zero; use the T flavor of `SMUL` when the data is in the top instead. The result is stored in R2, or in {R3, R2} for double-precision. Fractional operations (Q15/Q31) double the result using saturated adds to prevent overflow when multiplying -1×-1.

Table 6.16 **Multiply and multiply-accumulate instructions**

Instruction	Function	Description
Ordinary 32-bit multiplication works for both signed and unsigned		
`MUL`	`32 = 32 × 32`	Multiply
`MLA`	`32 = 32 + 32 × 32`	Multiply-accumulate
`MLS`	`32 = 32 – 32 × 32`	Multiply-subtract
unsigned long long = unsigned long × unsigned long		
`UMULL`	`64 = 32 × 32`	Unsigned multiply long
`UMLAL`	`64 = 64 + 32 × 32`	Unsigned multiply-accumulate long
`UMAAL`	`64 = 32 + 32 × 32 + 32`	Unsigned multiply-accumulate-add long
long long = long × long		
`SMULL`	`64 = 32 × 32`	Signed multiply long
`SMLAL`	`64 = 64 + 32 × 32`	Signed multiply-accumulate long
Packed arithmetic: short × short		
`SMUL{BB/BT/TB/TT}`	`32 = 16 × 16`	Signed multiply {bottom/top}
`SMLA{BB/BT/TB/TT}`	`32 = 32 + 16 × 16`	Signed multiply-accumulate {bottom/top}
`SMLAL{BB/BT/TB/TT}`	`64 = 64 + 16 × 16`	Signed multiply-accumulate long {bottom/top}
Fractional multiplication (Q31 / Q15)		
`SMULW{B/T}`	`32 = (32 × 16) >> 16`	Signed multiply word-halfword {bottom/top}
`SMLAW{B/T}`	`32 = 32 + (32 × 16) >> 16`	Signed multiply-add word-halfword {bottom/top}
`SMMUL{R}`	`32 = (32 × 32) >> 32`	Signed MSW multiply {round}
`SMMLA{R}`	`32 = 32 + (32 × 32) >> 32`	Signed MSW multiply-accumulate {round}
`SMMLS{R}`	`32 = 32 – (32 × 32) >> 32`	Signed MSW multiply-subtract {round}
long or long long = short × short + short × short		
`SMUAD`	`32 = 16 × 16 + 16 × 16`	Signed dual multiply-add
`SMUSD`	`32 = 16 × 16 – 16 × 16`	Signed dual multiply-subtract
`SMLAD`	`32 = 32 + 16 × 16 + 16 × 16`	Signed multiply-accumulate dual
`SMLSD`	`32 = 32 + 16 × 16 – 16 × 16`	Signed multiply-subtract dual
`SMLALD`	`64 = 64 + 16 × 16 + 16 × 16`	Signed multiply-accumulate long dual
`SMLSLD`	`64 = 64 + 16 × 16 – 16 × 16`	Signed multiply-subtract long dual

Table 6.17 Multiply and MAC code for various data types

First Operand (R0)	Second Operand (R1)	Product (R3/R2)	Multiply	MAC

short	short	short	`SMULBB R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`SMLABB R2, R0, R1`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

short	short	long	`SMULBB R2, R0, R1`	`SMLABB R2, R0, R1, R2`

short	short	long long	`MOV R2, #0`
			`MOV R3, #0`
			`SMLALBB R2, R3, R0, R1`
				`SMLALBB R2, R3, R0, R1`

long	short	long	`SMULWB R2, R0, R1`	`SMLAWB R2, R0, R1, R2`
long	long	long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
long	long	long long	`SMULL R2, R3, R0, R1`	`SMLAL R2, R3, R0, R1`

unsigned short	unsigned short	unsigned short	`MUL R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`MLA R2, R0, R1, R2`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

unsigned short	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long long	`UMULL R2, R3, R0, R1`	`UMLAL R2, R3, R0, R1`

Q15	Q15	Q15	`SMULBB R2, R0, R1`
			`QADD R2, R2, R2`
			`LSR R2, R2, #16`
				`SMLABB R2, R0, R1, R2`
				`SSAT R2, 16, R2`

Q15	Q15	Q31	`SMULBB R2, R0, R1` `QADD R2, R2, R2`	`SMULBB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q15	Q31	`SMULWB R2, R0, R1` `QADD R2, R2, R2`	`SMULWB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q31	Q31	`SMMUL R2, R0, R1` `QADD R2, R2, R2`	`SMMUL R3, R0, R1` `QDADD R2, R2, R3`

6.7.3 Floating-Point Instructions

Floating-point is more flexible than the fixed-point numbers favored in DSP and makes programming easier. Floating-point is widely used in graphics, scientific applications, and control algorithms. Floating-point arithmetic can be performed with a series of ordinary data-processing instructions but is faster and consumes less power using dedicated floating-point instructions and hardware.

The ARMv5 instruction set includes optional floating-point instructions. These instructions access at least 16 64-bit double-precision registers separate from the ordinary registers. These registers can also be treated as pairs of 32-bit single-precision registers. The registers are named D0–D15 as double-precision or S0–S31 as single-precision. For example, `VADD.F32 S2, S0, S1` and `VADD.F64 D2, D0, D1` perform single and double-precision floating-point adds, respectively. Floating-point instructions, listed in Table 6.18, are suffixed with `.F32` or `.F64` to indicate single- or double-precision floating-point.

Table 6.18 **ARM floating-point instructions**

Instruction	Function
`VABS Rd, Rm`	Rd = |Rm|
`VADD Rd, Rn, Rm`	Rd = Rn + Rm
`VCMP Rd, Rm`	Compare and set floating-point status flags
`VCVT Rd, Rm`	Convert between int and float
`VDIV Rd, Rn, Rm`	Rd = Rn / Rm
`VMLA Rd, Rn, Rm`	Rd = Rd + Rn * Rm
`VMLS Rd, Rn, Rm`	Rd = Rd – Rn * Rm
`VMOV Rd, Rm or #const`	Rd = Rm or constant
`VMUL Rd, Rn, Rm`	Rd = Rn * Rm
`VNEG Rd, Rm`	Rd = –Rm
`VNMLA Rd, Rn, Rm`	Rd = –(Rd + Rn * Rm)
`VNMLS Rd, Rn, Rm`	Rd = –(Rd – Rn * Rm)
`VNMUL Rd, Rn, Rm`	Rd = –Rn * Rm
`VSQRT Rd, Rm`	Rd = sqrt(Rm)
`VSUB Rd, Rn, Rm`	Rd = Rn – Rm

The `MRC` and `MCR` instructions are used to transfer data between the ordinary registers and the floating-point coprocessor registers.

ARM defines the Floating-Point Status and Control Register (FPSCR). Like the ordinary status register, it holds *N*, *Z*, *C*, and *V* flags for floating-point operations. It also specifies rounding modes, exceptions, and special conditions such as overflow, underflow, and divide-by-zero. The `VMRS` and `VMSR` instructions transfer information between a regular register and the FPSCR.

6.7.4 Power-Saving and Security Instructions

Battery-powered devices save power by spending most of their time in sleep mode. ARMv6K introduced instructions to support such power savings. The wait for interrupt (`WFI`) instruction allows the processor to enter a low-power state until an interrupt occurs. The system may generate interrupts based on user events (such as touching a screen) or on a periodic timer. The wait for event (`WFE`) instruction is similar but is helpful in multiprocessor systems (see Section 7.7.8) so that a processor can go to sleep until notified by another processor. It wakes up either during an interrupt or when another processor sends an event using the `SEV` instruction.

ARMv7 enhances the exception handling to support virtualization and security. In *virtualization*, multiple operating systems can run concurrently on the same processor, unaware of each other's existence. A hypervisor switches between the operating systems. The hypervisor operates at privilege level PL2. It is invoked with a hypervisor trap exception. With *security* extensions, the processor defines a secure state with limited means of entry and restricted access to secure portions of memory. Even if an attacker compromises the operating system, the secure kernel may resist tampering. For example, the secure kernel may be used to disable a stolen phone or to enforce digital rights management such that a user can't duplicate copyrighted content.

6.7.5 SIMD Instructions

The term SIMD (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. Moreover, when the components from 16 adjacent pixels are packed into a 128-bit quadword, the processing can be performed 16 times faster. Similarly, coordinates in a 3-dimensional graphics space are generally represented with 32-bit (single-precision) floating-point numbers. Four of these coordinates can be packed into a 128-bit quadword.

Most modern architectures offer SIMD arithmetic operations with wide SIMD registers packing multiple narrower operands. For example, the ARMv7 Advanced SIMD instructions share the registers from the floating-point unit. Moreover, these registers can also be paired to act as eight 128-bit quad words Q0–Q7. The registers pack together several 8-, 16-, 32-, or 64-bit integer or floating-point values. The instructions are suffixed with `.I8`, `.I16`, `.I32`, `.I64`, `.F32`, or `.F64` to indicate how the registers should be treated.

Figure 6.34 shows the `VADD.I8 D2, D1, D0` vector add instruction operating on eight pairs of 8-bit integers packed into 64-bit double words. Similarly `VADD.I32 Q2, Q1, Q0` adds four pairs of 32-bit integers packed into 128-bit quad words and `VADD.F32, D2, D1, D0` adds two pairs of 32-bit single-precision floating-point numbers packed into 64-bit double words. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ must not affect the result of $a_1 + b_1$.

Advanced SIMD instructions begin with `V`. They include the following categories:

- Basic arithmetic functions also defined for floating-point
- Loads and stores of multiple elements, including deinterleaving and interleaving
- Bitwise logical operations
- Comparisons
- Many flavors of shifts, additions, and subtractions with and without saturation
- Many flavors of multiply and MAC
- Miscellaneous instructions

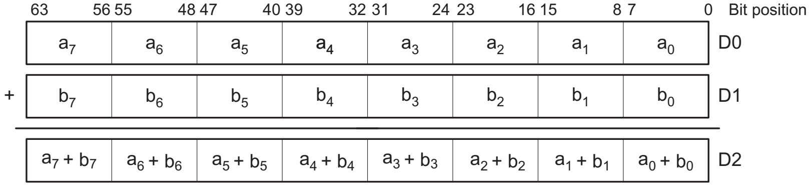

63
56 55
48 47
40 39
32 31
24 23
16 15
8 7
0
Bit position
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
D0
+
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
D1
$a_7 + b_7$ $a_6 + b_6$ $a_5 + b_5$ $a_4 + b_4$ $a_3 + b_3$ $a_2 + b_2$ $a_1 + b_1$ $a_0 + b_0$
D2

Figure 6.34 **Packed arithmetic: eight simultaneous 8-bit additions**

ARMv6 also defined a more limited set of SIMD instructions operating on the regular 32-bit registers. These include 8- and 16-bit addition and subtraction, and instructions to efficiently pack and unpack bytes and halfwords into a word. These instructions are useful to manipulate 16-bit data in DSP code.

6.7.6 64-bit Architecture

32-bit architectures allow a program to directly access at most 2^{32} bytes = 4 GB of memory. Large computer servers led the transition to 64-bit architectures that can access vast amounts of memory. Personal computers and then mobile devices followed. 64-bit architectures can sometimes be faster as well because they move more information with a single instruction.

Many architectures simply extend their general-purpose registers from 32 to 64 bits, but ARMv8 introduced a new instruction set as well to streamline idiosyncrasies. The classic instruction set lacks enough general-purpose registers for complex programs, forcing costly movement of data between registers and memory. Keeping the PC in R15 and SP in R13 also complicates the processor implementation, and programs often need a register containing the value 0.

The ARMv8 instructions are still 32 bits long and the instruction set looks very much like ARMv7, but with some problems cleaned up. In ARMv8, the register file is expanded to 31 64-bit registers (called X0–X30) and the PC and SP are no longer part of the general-purpose registers. X30 serves as the link register. Note that there is no X31 register; instead, it is called the zero register (ZR) and is hardwired to 0. Data-processing instructions can operate on 32- or 64-bit values, whereas loads and stores always use 64-bit addresses. To make room for the extra bits to specify source and destination registers, the condition field is removed from most instructions. However, branches can still be conditional. ARMv8 also streamlines exception handling, doubles the number of advanced SIMD registers, and adds instructions for AES and SHA cryptography. The instruction encodings are rather complex and do not classify into a handful of categories.

On reset, ARMv8 processors boot in 64-bit mode. The processor can drop into 32-bit mode by setting a bit in a system register and invoking an exception. It returns to 64-bit mode when the exception returns.

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

Almost all personal computers today use x86 architecture microprocessors. x86, also called IA-32, is a 32-bit architecture originally developed by Intel. AMD also sells x86 compatible microprocessors.

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than ARM. However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

x86 is an example of a Complex Instruction Set Computer (CISC) architecture. In contrast to RISC architectures such as ARM, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact, so as to save memory, when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer, but rather to illustrate some of the similarities and differences between x86 and ARM. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and ARM are summarized in Table 6.19.

Table 6.19 **Major differences between ARM and x86**

Feature	ARM	x86
# of registers	15 general purpose	8, some restrictions on purpose
# of operands	3–4 (2–3 sources, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition flags	yes	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

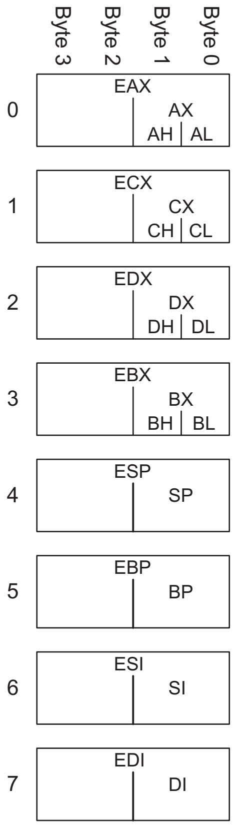


Figure 6.35 **x86 registers**

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to 32 bits. These registers are called EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.35.

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like SP in ARM, ESP is normally reserved for the stack pointer.

The x86 program counter is called the EIP (the *extended instruction pointer*). Like the ARM PC, it advances from one instruction to the next or can be changed with branch and function call instructions.

6.8.2 x86 Operands

ARM instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, x86 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

ARM instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. Table 6.20 lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Table 6.20 **Operand locations**

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <– EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <– EAX + 42`
register	memory	`add EAX, [20]`	`EAX <– EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <– Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <– Mem[20] + 42`

Table 6.21 Memory addressing modes

Example	Meaning	Comment
`add EAX, [20]`	`EAX <– EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <– EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <– EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <– EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <– EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.22 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <– AH + BL`	8-bit
`add AX, −1`	`AX <– AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <– EAX + EDX`	32-bit

Like ARM, x86 has a 32-bit memory space that is byte-addressable. However, x86 supports a wider variety of memory indexing modes. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.21 illustrates these combinations. The displacement can be an 8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the ARM base addressing mode for loads and stores. Like ARM, x86 also provides a scaled index. In x86, the scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address.

While ARM always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. Table 6.22 illustrates these variations.

6.8.3 Status Flags

x86, like many CISC architectures, uses condition flags (also called *status flags*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called EFLAGS, that stores the status flags. Some of the bits of the EFLAGS register are given in Table 6.23. Other bits are used by the operating system.

ARM's use of condition flags sets it apart from other RISC architectures.

Table 6.23 Selected EFLAGS

Name	Meaning
`CF` (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic
`ZF` (Zero Flag)	Result of last operation was zero
`SF` (Sign Flag)	Result of last operation was negative (msb = 1)
`OF` (Overflow Flag)	Overflow of two's complement arithmetic

The architectural state of an x86 processor includes EFLAGS as well as the eight registers and the EIP.

6.8.4 x86 Instructions

x86 has a larger set of instructions than ARM. Table 6.24 describes some of the general purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. `D` indicates the destination (a register or memory location), and `S` indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32×32-bit multiplication always takes one of the sources from EAX and always puts the 64-bit result in EDX and EAX. `LOOP` always stores the loop counter in ECX. `PUSH`, `POP`, `CALL`, and `RET` use the stack pointer, ESP.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, `JZ` jumps if the zero flag (*ZF*) is 1. `JNZ` jumps if the zero flag is 0. Like ARM, the jumps usually follow an instruction, such as the compare instruction (`CMP`), that sets the flags. Table 6.25 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike ARMv4, whose instructions are uniformly 32 bits, x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.36.[1]

---

[1] It is possible to construct 17-byte instructions if all the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

Table 6.24 **Selected x86 instructions**

Instruction	Meaning	Function
`ADD/SUB`	add/subtract	`D = D + S / D = D − S`
`ADDC`	add with carry	`D = D + S + CF`
`INC/DEC`	increment/decrement	`D = D + 1 / D = D − 1`
`CMP`	compare	Set flags based on `D − S`
`NEG`	negate	`D = − D`
`AND/OR/XOR`	logical AND/OR/XOR	`D = D op S`
`NOT`	logical NOT	$D = \overline{D}$
`IMUL/MUL`	signed/unsigned multiply	`EDX:EAX = EAX × D`
`IDIV/DIV`	signed/unsigned divide	`EDX:EAX/D` `EAX` = Quotient; `EDX` = Remainder
`SAR/SHR`	arithmetic/logical shift right	`D = D >>> S / D = D >> S`
`SAL/SHL`	left shift	`D = D << S`
`ROR/ROL`	rotate right/left	Rotate `D` by `S`
`RCR/RCL`	rotate right/left with carry	Rotate `CF` and `D` by `S`
`BT`	bit test	`CF = D[S]` (the `S`*th* bit of `D`)
`BTR/BTS`	bit test and reset/set	`CF = D[S]; D[S] = 0 / 1`
`TEST`	set flags based on masked bits	Set flags based on `D AND S`
`MOV`	move	`D = S`
`PUSH`	push onto stack	`ESP = ESP −4; Mem[ESP] = S`
`POP`	pop off stack	`D = MEM[ESP]; ESP = ESP + 4`
`CLC, STC`	clear/set carry flag	`CF = 0 / 1`
`JMP`	unconditional jump	relative jump: `EIP = EIP + S` absolute jump: `EIP = S`
`Jcc`	conditional jump	`if (flag) EIP = EIP + S`
`LOOP`	loop	`ECX = ECX −1` `if (ECX ≠ 0) EIP = EIP + imm`
`CALL`	function call	`ESP = ESP −4;` `MEM[ESP] = EIP; EIP = S`
`RET`	function return	`EIP = MEM[ESP]; ESP = ESP + 4`

Table 6.25 Selected branch conditions

Instruction	Meaning	Function after `CMP D, S`
`JZ/JE`	jump if `ZF` = 1	jump if `D` = `S`
`JNZ/JNE`	jump if `ZF` = 0	jump if `D` ≠ `S`
`JGE`	jump if `SF` = `OF`	jump if `D` ≥ `S`
`JG`	jump if `SF` = `OF` and `ZF` = 0	jump if `D` >`S`
`JLE`	jump if `SF` ≠ `OF` or `ZF` = 1	jump if `D` ≤ `S`
`JL`	jump if `SF` ≠ `OF`	jump if `D` <`S`
`JC/JB`	jump if `CF` = 1	
`JNC`	jump if `CF` = 0	
`JO`	jump if `OF` = 1	
`JNO`	jump if `OF` = 0	
`JS`	jump if `SF` = 1	
`JNS`	jump if `SF` = 0	

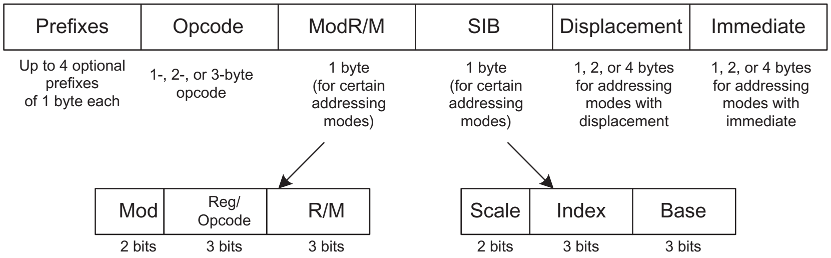

Prefixes
Opcode
ModR/M
SIB
Displacement
Immediate
Up to 4 optional prefixes of 1 byte each
1-, 2-, or 3-byte opcode
1 byte (for certain addressing modes)
1 byte (for certain addressing modes)
1, 2, or 4 bytes for addressing modes with displacement
1, 2, or 4 bytes for addressing modes with immediate
Mod
Reg/ Opcode
R/M
2 bits
3 bits
3 bits
Scale
Index
Base
2 bits
3 bits
3 bits

Figure 6.36 x86 instruction encodings

The *opcode* may be 1, 2, or 3 bytes. It is followed by four optional fields: *ModR/M*, *SIB*, *Displacement*, and *Immediate*. *ModR/M* specifies an addressing mode. *SIB* specifies the scale, index, and base registers in certain addressing modes. *Displacement* indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. And *Immediate* is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The *ModR/M* byte uses the 2-bit *Mod* and 3-bit *R/M* field to specify the addressing mode for one of the operands. The operand can come from one of the eight registers, or from one of 24 memory addressing modes. Due to artifacts in the encodings, the ESP and EBP registers are not available for use as the base or index register in certain addressing modes. The *Reg* field specifies the register used as the other operand. For certain instructions that do not require a second operand, the *Reg* field is used to specify three more bits of the *opcode*.

In addressing modes using a scaled index register, the *SIB* byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the *SIB* byte also specifies the base register.

ARM fully specifies the instruction in the *cond*, *op*, and *funct* fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `add AL, imm8` performs an 8-bit add of an immediate to AL. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The A register (AL, AX, or EAX) is called the *accumulator*. On the other hand, `add D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, *D* (memory or a register). It is represented with the 1-byte *opcode* 0x80 followed by one or more bytes specifying *D*, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the *opcode* specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form. Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specifies which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the *opcode* to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the *prefix* 0x66 appears before the *opcode*, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid 1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, IA-64 has yet to become a market success. Most computers needing the large address space now use the 64-bit extensions of x86.

x86 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 *prefix* is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, x86 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000's it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those curious about more details of the x86 architecture, the x86 Intel Architecture Software Developer's Manual is freely available on Intel's Web site.

6.8.7 The Big Picture

ARM strikes a balance between simple instructions and dense code by including features such as condition flags and shifted register operands. Thease features make ARM code more compact than other RISC architectures.

This section has given a taste of some of the differences between the ARM RISC architecture and the x86 CISC architecture. x86 tends to have shorter programs, because a complex instruction is equivalent to a series of simple ARM instructions and because the instructions are encoded to minimize memory use. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs. It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs, because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are:

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

ARM is a 32-bit architecture because it operates on 32-bit data. The ARM architecture has 16 registers which include 15 general-purpose registers and the PC. In principle, any of the general-purpose registers can be used in any code. However, by convention, certain registers are reserved for certain purposes for ease of programming and so that functions written by different programmers can communicate easily. For example, R14 (the link register LR) holds the return address after a `BL` instruction, and R0–R3 hold the arguments of a function. ARM has a byte-addressable memory system with 32-bit addresses. Instructions are 32 bits long and are word-aligned for efficient access. This chapter discussed the most commonly used ARM instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel Pentium processor in 1993 will generally still run (and run much faster) on the Intel Xeon or AMD Phenom processors in 2015.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture. Microarchitecture is the link between hardware and software engineering. And, we believe it is one of the most exciting topics in all of engineering: You will learn to build your own microprocessor!

Exercises

Exercise 6.1 Give three examples from the ARM architecture of each of the architecture design principles: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Exercise 6.2 The ARM architecture has a register set that consists of 16 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are advantages and disadvantages of this architecture over the ARM architecture?

Exercise 6.3 Consider memory storage of a 32-bit word stored at memory word 42 in a byte-addressable memory.

(a) What is the byte address of memory word 42?

(b) What are the byte addresses that memory word 42 spans?

(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Clearly label the byte address corresponding to each data byte value.

Exercise 6.4 Repeat Exercise 6.3 for memory storage of a 32-bit word stored at memory word 15 in a byte-addressable memory.

Exercise 6.5 Explain how the following ARM program can be used to determine whether a computer is big-endian or little-endian:

```
MOV  R0, #100
LDR  R1, =0xABCD876     ; R1 = 0xABCD876
STR  R1, [R0]
LDRB R2, [R0, #1]
```

Exercise 6.6 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) SOS

(b) Cool

(c) boo!

Exercise 6.7 Repeat Exercise 6.6 for the following strings.

(a) howdy

(b) lions

(c) To the rescue!

Exercise 6.8 Show how the strings in Exercise 6.6 are stored in a byte-addressable memory on a little-endian machine starting at memory address 0x00001050C. Clearly indicate the memory address of each byte.

Exercise 6.9 Repeat Exercise 6.8 for the strings in Exercise 6.7.

Exercise 6.10 Convert the following ARM assembly code into machine language. Write the instructions in hexadecimal.

```
MOV R10, #63488
LSL R9, R6, #7
STR R4, [R11, R8]
ASR R6, R7, R3
```

Exercise 6.11 Repeat Exercise 6.10 for the following ARM assembly code:

```
ADD R8, R0, R1
LDR R11, [R3, #4]
SUB R5, R7, #0x58
LSL R3, R2, #14
```

Exercise 6.12 Consider data-processing instructions with an immediate *Src2*.

(a) Which instructions from Exercise 6.10 are in this format?

(b) Write out the 12-bit immediate field (*imm12*) of the instructions from part (a), then write them as 32-bit immediates.

Exercise 6.13 Repeat Exercise 6.12 for the instructions in Exercise 6.11.

Exercise 6.14 Convert the following program from machine language into ARM assembly language. The numbers on the left are the instruction addresses in memory, and the numbers on the right give the instruction at that address. Then reverse engineer a high-level program that would compile into this assembly language routine and write it. Explain in words what the program does. R0 and R1 are the input, and they initially contain positive numbers, `a` and `b`. At the end of the program, R0 is the output.

```
0x00008008  0xE3A02000
0x0000800C  0xE1A03001
0x00008010  0xE1510000
0x00008014  0x8A000002
0x00008018  0xE2822001
0x0000801C  0xE0811003
0x00008020  0xEAFFFFFA
0x00008024  0xE1A00002
```

Exercise 6.15 Repeat Exercise 6.14 for the following machine code. R0 and R1 are the inputs. R0 contains a 32-bit number and R1 is the address of a 32-element array of characters (char).

```
0x00008104  0xE3A0201F
0x00008108  0xE1A03230
0x0000810C  0xE2033001
0x00008110  0xE4C13001
0x00008114  0xE2522001
0x00008118  0x5AFFFFFA
0x0000811C  0xE1A0F00E
```

Exercise 6.16 The `NOR` instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: R0 = R1 NOR R2. Use as few instructions as possible.

Exercise 6.17 The `NAND` instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: R0 = R1 NAND R2. Use as few instructions as possible.

Exercise 6.18 Consider the following high-level code snippets. Assume the (signed) integer variables `g` and `h` are in registers R0 and R1, respectively.

(i)

```
if (g >= h)
  g = g + h;
else
  g = g − h;
```

(ii)

```
if (g < h)
  h = h + 1;
else
  h = h * 2;
```

(a) Write the code snippets in ARM assembly language assuming conditional execution is available for branch instructions only. Use as few instructions as possible (within these parameters).

(b) Write the code snippets in ARM assembly language with conditional execution available for all instructions. Use as few instructions as possible.

(c) Compare the difference in code density (i.e., number of instructions) between (a) and (b) for each code snippet and discuss any advantages or disadvantages.

Exercise 6.19 Repeat Exercise 6.18 for the following code snippets.

(i)
```
if (g > h)
  g = g + 1;
else
  h = h − 1;
```

(ii)
```
if (g <= h)
  g = 0;
else
  h = 0;
```

Exercise 6.20 Consider the following high-level code snippet. Assume that the base addresses of `array1` and `array2` are held in R1 and R2 and that `array2` is initialized before it is used.

```
int i;
int array1[100];
int array2[100];
...
for (i=0; i<100; i=i+1)
    array1[i] = array2[i];
```

(a) Write the code snippet in ARM assembly without using pre- or post-indexing or a scaled register. Use as few instructions as possible (given the constraints).

(b) Write the code snippet in ARM assembly with pre- or post-indexing and a scaled register available. Use as few instructions as possible.

(c) Compare the difference in code density (i.e., number of instructions) between (a) and (b). Discuss any advantages or disadvantages.

Exercise 6.21 Repeat Exercise 6.20 for the following high-level code snippet. Assume that `temp` is initialized before it is used and that R3 holds the base address of `temp`.

```
int i;
int temp[100];
...
for (i=0; i<100; i=i+1)
  temp[i] = temp[i] * 128;
```

Exercise 6.22 Consider the following two code snippets. Assume R1 holds `i` and that R0 holds the base address of the `vals` array.

(i)
```
int i;
int vals[200];

for (i=0; i < 200; i=i+1)
  vals[i] = i;
```

(ii)

```
int i;
int vals[200];

for (i=199; i >= 0; i = i-1)
  vals[i] = i;
```

(a) Are the code snippets functionally equivalent?

(b) Write each code snippet using ARM assembly language. Use as few instructions as possible.

(c) Discuss any advantages or disadvantages of one construct over the other.

Exercise 6.23 Repeat Exercise 6.22 for the following high-level code snippets. Assume R1 holds `i`, R0 holds the base address of the `nums` array, and that the array is initialized before use.

(i)

```
int i;
int nums[10];
...
for (i=0; i < 10; i=i+1)
  nums[i] = nums[i]/2;
```

(ii)

```
int i;
int nums[10];
...
for (i=9; i >= 0; i = i-1)
  nums[i] = nums[i]/2;
```

Exercise 6.24 Write a function in a high-level language for `int find42(int array[], int size)`. `size` specifies the number of elements in `array`, and `array` specifies the base address of the `array`. The function should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value –1.

Exercise 6.25 The high-level function `strcpy` copies the character string `src` to the character string `dst`.

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Implement the `strcpy` function in ARM assembly code. Use R4 for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` function call.

Assume SP = 0xBEFFF000 just before `strcpy` is called.

This simple string copy function has a serious flaw: it has no way of knowing that `dst` has enough space to receive `src`. If a malicious programmer were able to execute `strcpy` with a long string `src`, the programmer might be able to write bytes all over memory, possibly even modifying code stored in subsequent memory locations. With some cleverness, the modified code might take over the machine. This is called a buffer overflow attack; it is employed by several nasty programs, including the infamous Blaster worm, which caused an estimated $525 million in damages in 2003.

Exercise 6.26 Convert the high-level function from Exercise 6.24 into ARM assembly code.

Exercise 6.27 Consider the ARM assembly code below. `func1`, `func2`, and `func3` are non-leaf functions. `func4` is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function.

```
0x00091000 func1 ...   ; func1 uses R4-R10
0x00091020 BL func2
. . .
0x00091100 func2 ...   ; func2 uses R0-R5
0x0009117C BL func3
. . .
0x00091400 func3 ...   ; func3 uses R3, R7-R9
0x00091704 BL func4
. . .
0x00093008 func4 ...   ; func4 uses R11-R12
0x00093118 MOV PC, LR
```

(a) How many words are the stack frames of each function?

(b) Sketch the stack after `func4` is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Give values where possible.

Exercise 6.28 Each number in the Fibonacci series is the sum of the previous two numbers. Table 6.26 lists the first few numbers in the series, *fib*(*n*).

(a) What is *fib*(*n*) for $n = 0$ and $n = -1$?

(b) Write a function called `fib` in a high-level language that returns the Fibonacci number for any nonnegative value of *n*. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into ARM assembly code. Add comments after every line of code that explain clearly what it does. Use the Keil MDK-ARM simulator to test your code on *fib*(9). (See the Preface for how to install the Keil MDK-ARM simulator.)

Table 6.26 **Fibonacci series**

n	1	2	3	4	5	6	7	8	9	10	11	…
fib(*n*)	1	1	2	3	5	8	13	21	34	55	89	…

Exercise 6.29 Consider Code Example 6.27. For this exercise, assume *factorial*(*n*) is called with input argument $n = 5$.

(a) What value is in R0 when `factorial` returns to the calling function?

(b) Suppose you replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {R0, R1}` and `POP {R1, R2}`, respectively. Will the program:
   (1) enter an infinite loop but not crash;
   (2) crash (cause the stack to grow or shrink beyond the dynamic data segment or the PC to jump to a location outside the program);
   (3) produce an incorrect value in R0 when the program returns to loop (if so, what value?); or
   (4) run correctly despite the deleted lines?

(c) Repeat part (b) with the following instruction modifications:
   (i) replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {R3, LR}` and `POP {R3, LR}`, respectively.
   (ii) replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {LR}` and `POP {LR}`, respectively.
   (iii) delete the instruction at address 0x8510.

Exercise 6.30 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative b. He goes overboard in the use of function calls and recursion and produces the following high-level code for functions `f` and `g`.

```
// high-level code for functions f and g
int f(int a, int b) {
  int j;

  j = a;

  return j + a + g(b);
}
int g(int x) {
  int k;
  k = 3;

  if (x == 0) return 0;
  else return k + g(x − 1);
}
```

Ben then translates the two functions into assembly language as follows. He also writes a function, `test`, that calls the function `f(5, 3)`.

```
; ARM assembly code
; f: R0 = a, R1 = b, R4 = j;
; g: R0 = x, R4 = k

0x00008000 test   MOV  R0, #5             ; a = 5
0x00008004        MOV  R1, #3             ; b = 3
0x00008008        BL   f                  ; call f(5, 3)
0x0000800C loop   B    loop               ; and loop forever
0x00008010 f      PUSH {R1,R0,LR,R4}      ; save registers on stack
0x00008014        MOV  R4, R0             ; j = a
0x00008018        MOV  R0, R1             ; place b as argument for g
0x0000801C        BL   g                  ; call g(b)
0x00008020        MOV  R2, R0             ; place return value in R2
0x00008024        POP  {R1,R0}            ; restore a and b after call
0x00008028        ADD  R0, R2, R0         ; R0 = g(b) + a
0x0000802C        ADD  R0, R0, R4         ; R0 = (g(b) + a) + j
0x00008030        POP  {R4,LR}            ; restore R4, LR
0x00008034        MOV  PC, LR             ; return
0x00008038 g      PUSH {R4,LR}            ; save registers on stack
0x0000803C        MOV  R4, #3             ; k = 3
0x00008040        CMP  R0, #0             ; x == 0?
0x00008044        BNE  else               ; branch when not equal
0x00008048        MOV  R0, #0             ; if equal, return value = 0
0x0000804C        B    done               ; and clean up
0x00008050 else   SUB  R0, R0, #1         ; x = x - 1
0x00008054        BL   g                  ; call g(x - 1)
0x00008058        ADD  R0, R0, R4         ; R0 = g(x - 1) + k
0x0000805C done   POP  {R4,LR}            ; restore R0,R4,LR from stack
0x00008060        MOV  PC, LR             ; return
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.14 to help you answer the following questions.

(a) If the code runs starting at `test`, what value is in R0 when the program gets to `loop`? Does his program correctly compute $2a + 3b$?

(b) Suppose Ben changes the instructions at addresses 0x00008010 and 0x00008030 to `PUSH {R1,R0,R4}` and `POP {R4}`, respectively. Will the program

(1) enter an infinite loop but not crash;

(2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program);

(3) produce an incorrect value in R0 when the program returns to `loop` (if so, what value?), or

(4) run correctly despite the deleted lines?

(c) Repeat part (b) when the following instructions are changed. Note that labels aren't changed, only instructions.
  (i) instructions at 0x00008010 and 0x00008024 change to `PUSH {R1,LR, R4}` and `POP {R1}`, respectively.
  (ii) instructions at 0x00008010 and 0x00008024 change to `PUSH {R0,LR, R4}` and `POP {R0}`, respectively.
  (iii) instructions at 0x00008010 and 0x00008030 change to `PUSH {R1,R0, LR}` and `POP {LR}`, respectively.
  (iv) instructions at 0x00008010, 0x00008024, and 0x00008030 are deleted.
  (v) instructions at 0x00008038 and 0x0000805C change to `PUSH {R4}` and `POP {R4}`, respectively.
  (vi) instructions at 0x00008038 and 0x0000805C change to `PUSH {LR}` and `POP {LR}`, respectively.
  (vii) instructions at 0x00008038 and 0x0000805C are deleted.

Exercise 6.31 Convert the following branch instructions into machine code. Instruction addresses are given to the left of each instruction.

(a)
```
0x0000A000       BEQ LOOP
0x0000A004       ...
0x0000A008       ...
0x0000A00C LOOP  ...
```

(b)
```
0x00801000       BGE DONE
...
0x00802040 DONE  ...
```

(c)
```
0x0000B10C BACK  ...
...              ...
0x0000D000       BHI BACK
```

(d)
```
0x00103000       BL  FUNC
...              ...
0x0011147C FUNC  ...
```

(e)
```
0x00008004 L1    ...
...              ...
0x0000F00C       B L1
```

Exercise 6.32 Consider the following ARM assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x000A0028 FUNC1      MOV R4, R1
0x000A002C            ADD R5, R3, R5, LSR #2
0x000A0030            SUB R4, R0, R3, ROR R4
0x000A0034            BL  FUNC2
...                   ...
0x000A0038 FUNC2      LDR R2, [R0, #4]
0x000A003C            STR R2, [R1, -R2]
```

```
0x000A0040            CMP R3, #0
0x000A0044            BNE ELSE
0x000A0048            MOV PC, LR
0x000A004C ELSE       SUB R3, R3, #1
0x000A0050            B FUNC2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

(b) List the addressing mode used at each line of code.

Exercise 6.33 Consider the following C code snippet.

```
// C code
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1)
    array[i] = compare(num, i);
}
int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
  else
    return 0;
}
int sub(int a, int b) {
  return a − b;
}
```

(a) Implement the C code snippet in ARM assembly language. Use R4 to hold the variable `i`. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the `setArray` function (see the end of Section 6.3.7).

(b) Assume `setArray` is the first function called. Draw the status of the stack before calling `setArray` and during each function call. Indicate the names of registers and variables stored on the stack, mark the location of SP, and clearly mark each stack frame.

(c) How would your code function if you failed to store LR on the stack?

Exercise 6.34 Consider the following high-level function.

```
// C code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n − 1, k + 1);
  return b * k;
}
```

(a) Translate the high-level function `f` into ARM assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the ARM preserved register conventions. Clearly comment your code. You can use the ARM `MUL` instruction. The function starts at instruction address 0x00008100. Keep local variable `b` in R4.

(b) Step through your function from part (a) by hand for the case of `f(2, 4)`. Draw a picture of the stack similar to the one in Figure 6.14, and assume that SP is equal to 0xBFF00100 when `f` is called. Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (SP). Clearly mark each stack frame. You might also find it useful to keep track of the values in R0, R1, and R4 throughout execution. Assume that when `f` is called, R4 = 0xABCD and LR = 0x00008010. What is the final value of R0?

Exercise 6.35 Give an example of the worst case for a forward branch (i.e., a branch to a higher instruction address). The worst case is when the branch cannot branch far. Show instructions and instruction addresses.

Exercise 6.36 The following questions examine the limitations of the branch instruction, `B`. Give your answer in number of instructions relative to the branch instruction.

(a) In the worst case, how far can `B` branch forward (i.e., to higher addresses)? (The worst case is when the branch instruction cannot branch far.) Explain using words and examples, as needed.

(b) In the best case, how far can `B` branch forward? (The best case is when the branch instruction can branch the farthest.) Explain.

(c) In the worst case, how far can `B` branch backward (to lower addresses)? Explain.

(d) In the best case, how far can `B` branch backward? Explain.

Exercise 6.37 Explain why it is advantageous to have a large immediate field, *imm24*, in the machine format for the branch instructions, `B` and `BL`.

Exercise 6.38 Write assembly code that branches to an instruction 32 Minstructions from the first instruction. Recall that 1 Minstruction = 2^{20} instructions =1,048,576 instructions. Assume that your code begins at address 0x00008000. Use a minimum number of instructions.

Exercise 6.39 Write a function in high-level code that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. After writing the high-level code, convert it to ARM assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.40 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a function called `concat` that concatenates (joins together) the two strings: `void concat(char string1[], char string2[], char stringconcat[])`. The function does not return a value. It concatenates `string1` and `string2` and places the resulting string in `stringconcat`. You may assume that the character array `stringconcat` is large enough to accommodate the concatenated string.

(b) Convert the function from part (a) into ARM assembly language.

Exercise 6.41 Write an ARM assembly program that adds two positive single-precision floating point numbers held in R0 and R1. Do not use any of the ARM floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, etc.) or numbers that overflow or underflow. Use the Keil MDK-ARM simulator to test your code. (See the Preface for how to install the Keil MDK-ARM simulator.) You will need to manually set the values of R0 and R1 to test your code. Demonstrate that your code functions reliably.

Exercise 6.42 Consider the following ARM program. Assume the instructions are placed starting at memory address 0x8400 and that `L1` is at memory address 0x10024.

```
; ARM assembly code
MAIN
   PUSH {LR}
   LDR  R2, =L1  ; this is translated into a PC-relative load
   LDR  R0, [R2]
   LDR  R1, [R2, #4]
   BL   DIFF
   POP  {LR}
   MOV  PC, LR
DIFF
   SUB  R0, R0, R1
   MOV  PC, LR
   ...
  L1
```

(a) First show the instruction address next to each assembly instruction.

(b) Describe the symbol table: i.e., list the address of each of the labels.

(c) Convert all instructions into machine code.

(d) How big (how many bytes) are the data and text segments?

(e) Sketch a memory map showing where data and instructions are stored, similar to Figure 6.31.

Exercise 6.43 Repeat Exercise 6.42 for the following ARM code. Assume the instructions are placed starting at memory address 0x8534 and that `L2` is at memory address 0x1305C.

```
; ARM assembly code
MAIN
  PUSH  {R4,LR}
  MOV   R4, #15
  LDR   R3, =L2  ; this is translated into a PC-relative load
  STR   R4, [R3]
  MOV   R1, #27
  STR   R1, [R3, #4]
  LDR   R0, [R3]
  BL    GREATER
  POP   {R4,LR}
  MOV   PC, LR
GREATER
  CMP   R0, R1
  MOV   R0, #0
  MOVGT R0, #1
  MOV   PC, LR
  ...
L2
```

Exercise 6.44 Name two ARM instructions that can increase code density (i.e., decrease the number of instructions in a program). Give examples of each, showing equivalent ARM assembly code with and without using the instructions.

Exercise 6.45 Explain the advantages and disadvantages of conditional execution.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs (but are usually open to any assembly language).

Question 6.1 Write ARM assembly code for swapping the contents of two registers, R0 and R1. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write ARM assembly code that finds the subset of the array with the largest sum. Assume that the array's base address and the number of array elements are in R0 and R1, respectively. Your code should place the resulting subset of the array starting at the base address in R2. Write code that runs as fast as possible.

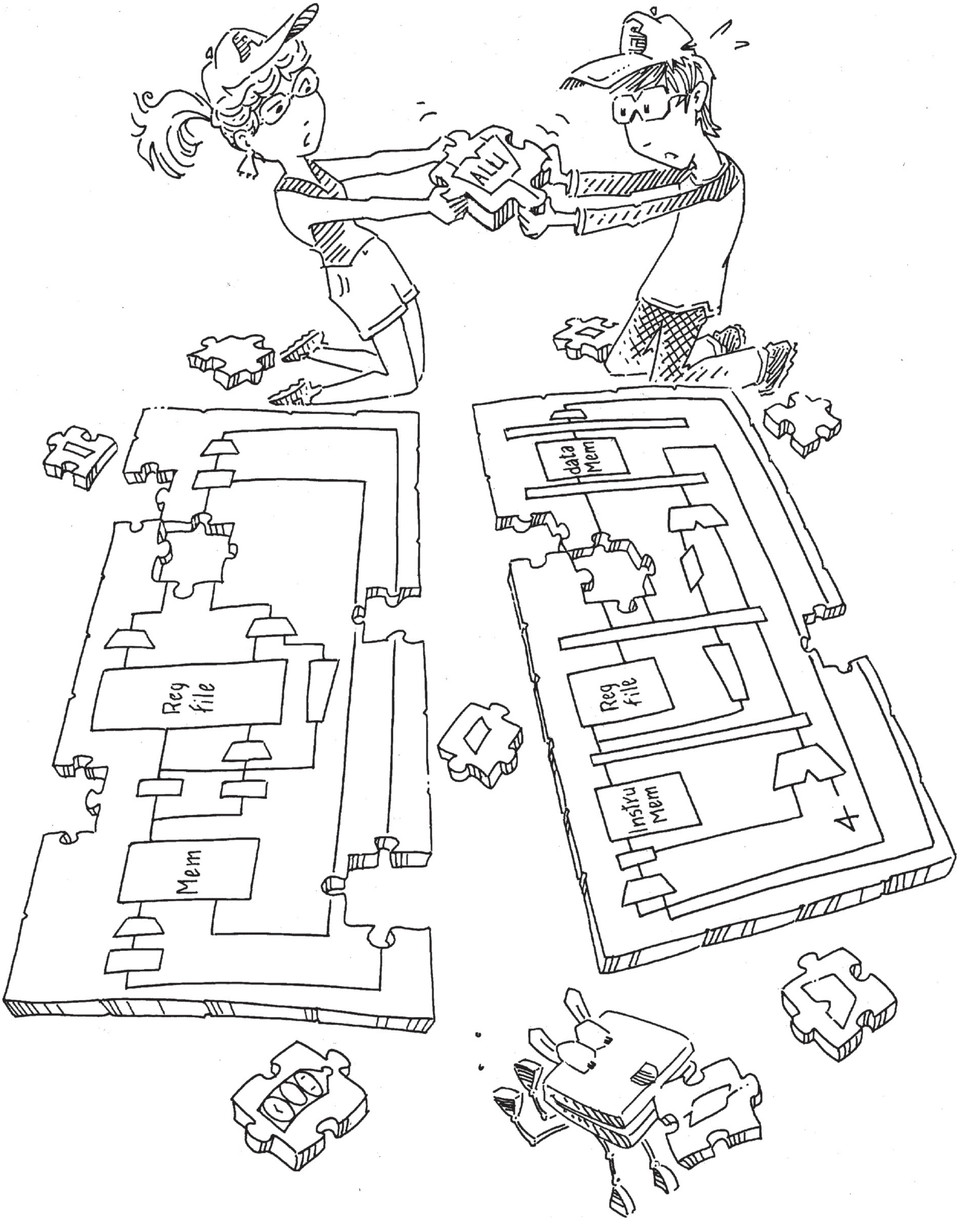
Question 6.3 You are given an array that holds a C string. The string forms a sentence. Design an algorithm for reversing the words in the sentence and storing the new sentence back in the array. Implement your algorithm using ARM assembly code.

Question 6.4 Design an algorithm for counting the number of 1's in a 32-bit number. Implement your algorithm using ARM assembly code.

Question 6.5 Write ARM assembly code to reverse the bits in a register. Use as few instructions as possible. Assume the register of interest is R3.

Question 6.6 Write ARM assembly code to test whether overflow occurs when R2 and R3 are added. Use a minimum number of instructions.

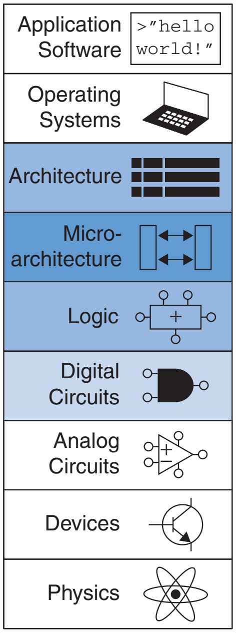
Question 6.7 Design an algorithm for testing whether a given string is a palindrome. (Recall that a palindrome is a word that is the same forward and backward. For example, the words "wow" and "racecar" are palindromes.) Implement your algorithm using ARM assembly code


ALU
data Mem
Reg file
Instru Mem
4
Reg file
Mem

Microarchitecture

7

7.1 Introduction
7.2 Performance Analysis
7.3 Single-Cycle Processor
7.4 Multicycle Processor
7.5 Pipelined Processor
7.6 HDL Representation*
7.7 Advanced Microarchitecture*
7.8 Real-World Perspective: Evolution of ARM Microarchitecture*
7.9 Summary
Exercises
Interview Questions



7.1 INTRODUCTION

In this chapter, you will learn how to piece together a microprocessor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward, and by this point you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory. And you have learned about the ARM architecture, which specifies the programmer's view of the ARM processor in terms of registers, instructions, and memory.

This chapter covers *microarchitecture*, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as ARM, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We design three different microarchitectures in this chapter to illustrate the trade-offs.

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and architectural state. The *architectural state* for the ARM processor consists of 16 32-bit registers and the status register. Any ARM microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some microarchitectures contain

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00007-8
© 2016 Elsevier Inc. All rights reserved.

additional *nonarchitectural state* to either simplify the logic or improve performance; we point this out as it arises.

To keep the microarchitectures easy to understand, we consider only a subset of the ARM instruction set. Specifically, we handle the following instructions:

- Data-processing instructions: `ADD`, `SUB`, `AND`, `ORR` (with register and immediate addressing modes but no shifts)
- Memory instructions: `LDR`, `STR` (with positive immediate offset)
- Branches: `B`

These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

7.1.2 Design Process

We divide our microarchitectures into two interacting parts: the *datapath* and the *control unit*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. We are implementing the 32-bit ARM architecture, so we use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter, registers, and status register). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the five state elements: the program counter, register file, status register, and instruction and data memories.

In Figure 7.1, heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 4-bit address busses on the register file. Narrow lines indicate 1-bit buses, and blue lines are used for control signals, such as the register file write enable. We use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

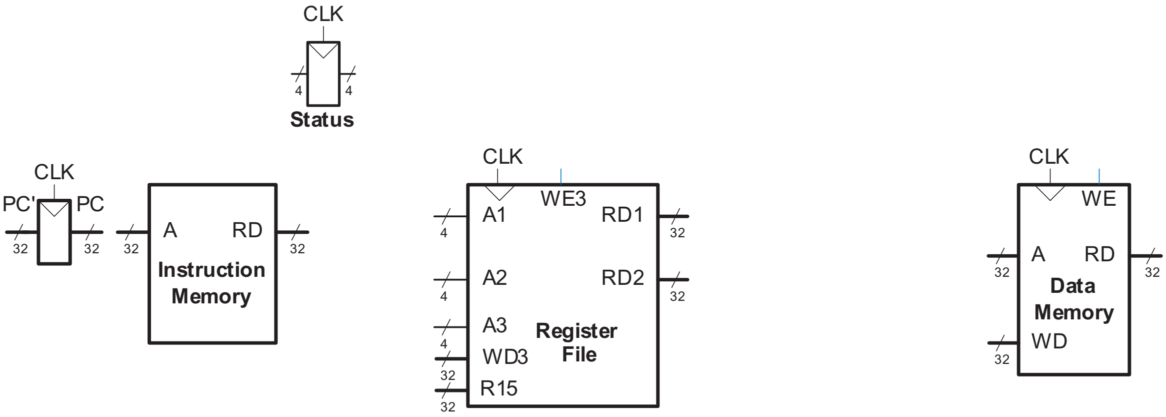


Figure 7.1 State elements of ARM processor

Although the *program counter* (PC) is logically part of the register file, it is read and written on every cycle independent of the normal register file operation and is more naturally built as a stand-alone 32-bit register. Its output, *PC*, points to the current instruction. Its input, *PC′*, indicates the address of the next instruction.

The *instruction memory* has a single read port.[1] It takes a 32-bit instruction address input, *A*, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.

The 15-element × 32-bit register file holds registers R0–R14 and has an additional input to receive R15 from the PC. The register file has two read ports and one write port. The read ports take 4-bit address inputs, *A1* and *A2*, each specifying one of $2^4 = 16$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 4-bit address input, *A3*; a 32-bit write data input, *WD3*; a write enable input, *WE3*; and a clock. If the write enable is asserted, then the register file writes the data into the specified register on the rising edge of the clock. A read of R15 returns the value from the PC plus 8, and writes to R15 must be specially handled to update the PC because it is separate from the register file.

The *data memory* has a single read/write port. If its write enable, *WE*, is asserted, then it writes data *WD* into address *A* on the rising edge of the clock. If its write enable is 0, then it reads address *A* onto *RD*.

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. ARM processors normally initialize the PC to `0x00000000` on reset, and we start our programs there.

Treating the PC as part of the register file complicates the system design, and complexity ultimately means more gates and higher power consumption. Most other architectures treat the PC as a special register that is only updated by branches, not by ordinary data-processing instructions. As described in Section 6.7.6, ARM's 64-bit ARMv8 architecture also makes the PC a special register separate from the register file.

---

[1] This is an oversimplification used to treat the instruction memory as a ROM; in most real processors, the instruction memory must be writable so that the OS can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a combined memory for instructions and data that can be both read and written.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, then the new data appears at *RD* after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must setup before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

Examples of classic multicycle processors include the 1947 MIT Whirlwind, the IBM System/360, the Digital Equipment Corporation VAX, the 6502 used in the Apple II, and the 8088 used in the IBM PC. Multicycle microarchitectures are still used in inexpensive microcontrollers such as the 8051, the 68HC11, and the PIC16-series found in appliances, toys, and gadgets.

7.1.3 Microarchitectures

In this chapter, we develop three microarchitectures for the ARM architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The *single-cycle microarchitecture* executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction. Moreover, the processor requires separate instruction and data memories, which is generally unrealistic.

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. The multicycle processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data. Therefore, multicycle processors were the historical choice for inexpensive systems.

Intel processors have been pipelined since the 80486 was introduced in 1989. Nearly all RISC microprocessors are also pipelined. ARM processors have been pipelined since the original ARM1 in 1985. A pipelined ARM Cortex-M0 requires only about 12,000 logic gates, so in a modern integrated circuit it is so small that one needs a microscope to see it and the manufacturing cost is a fraction of a penny. Combined with memory and peripherals, a commercial Cortex-M0 chip such as the Freescale Kinetis still costs less than 50 cents. Thus, pipelined processors are replacing their slower multicycle siblings in even the most cost-sensitive applications.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. Pipelined processors must access instructions and data in the same cycle; they generally use separate instruction and data caches for this purpose, as discussed in Chapter 8. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to achieve even more speed in modern high-performance microprocessors.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Precise cost calculations require detailed knowledge of the implementation technology but, in general, more gates and more memory mean more dollars.

This section lays the foundation for analyzing performance. There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real-world performance. For example, microprocessor makers often market their products based on the clock frequency and the number of cores. However, they gloss over the complications that some processors accomplish more work than others in a clock cycle and that this varies from program to program. What is a buyer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you have not written your program yet or if somebody else who does not have your program is making the measurements. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

Equation 7.1 gives the execution time of a program, measured in seconds.

$$Execution\ Time = \left(\#instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \tag{7.1}$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to

Dhrystone, CoreMark, and SPEC are three popular benchmarks. The first two are *synthetic benchmarks* comprising important common pieces of programs. Dhrystone was developed in 1984 and remains commonly used for embedded processors, although the code is somewhat unrepresentative of real-life programs. CoreMark is an improvement over Dhrystone and involves matrix multiplications that exercise the multiplier and adder, linked lists to exercise the memory system, state machines to exercise the branch logic, and cyclical redundancy checks that involve many parts of the processor. Both benchmarks are less than 16 KB in size and do not stress the instruction cache.

The SPEC CINT2006 benchmark from the Standard Performance Evaluation Corporation is composed of real programs, including h264ref (video compression), sjeng (an artificial intelligence chess player), hmmer (protein sequence analysis), and gcc (a C compiler). The benchmark is widely used for high-performance processors because it stresses the entire CPU in a representative way.

execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we assume that we are executing known programs on an ARM processor, so the number of instructions for each program is constant, independent of the microarchitecture. The *cycles per instruction* (*CPI*) is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (*instructions per cycle*, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we assume we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances have historically doubled transistor speeds every 4–6 years, so a microprocessor built today will be faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

Many other factors affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world does not help surfing the Internet on a dial-up connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

We first design a microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is performed by the datapath at any given time. The control unit contains combinational logic that generates the appropriate control signals based on the current instruction. We conclude by analyzing the performance of the single-cycle processor.

7.3.1 Single-Cycle Datapath

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray. The status register is part of the controller and will be omitted while we focus on the datapath.

The program counter contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*.

The processor's actions depend on the specific instruction that was fetched. First, we will work out the datapath connections for the `LDR` instruction with positive immediate offset. Then, we will consider how to generalize the datapath to handle other instructions.

LDR

For the `LDR` instruction, the next step is to read the source register containing the base address. This register is specified in the *Rn* field of the instruction, $Instr_{19:16}$. These bits of the instruction are connected to the address input of one of the register file ports, *A1*, as shown in Figure 7.3. The register file reads the register value onto *RD1*.

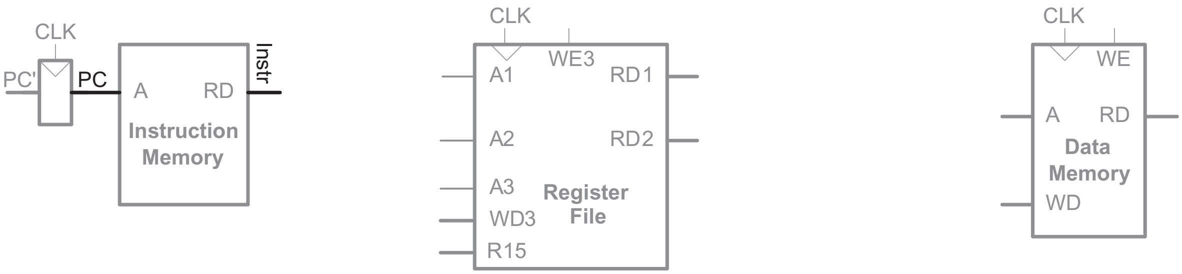

CLK
PC'
PC
A
RD
Instruction Memory
Instr
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
CLK
WE
A
RD
Data Memory
WD

Figure 7.2 Fetch instruction from memory

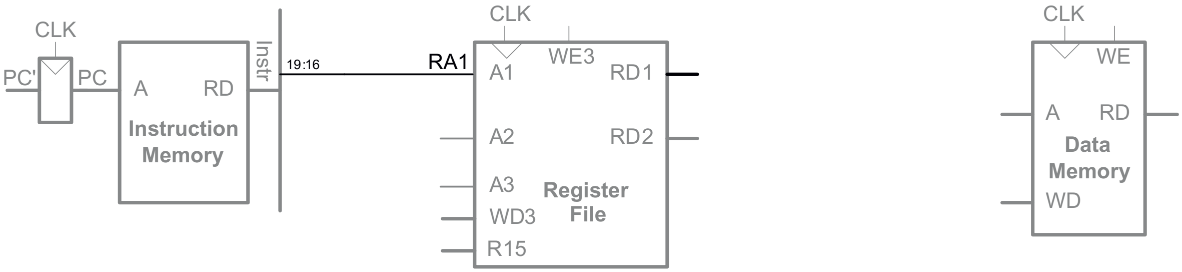

CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
CLK
WE
A
RD
Data Memory
WD

Figure 7.3 Read source operand from register file

The LDR instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{11:0}$. It is an unsigned value, so it must be zero-extended to 32 bits, as shown in Figure 7.4. The 32-bit value is called *ExtImm*. Zero extension simply means prepending leading zeros: $ImmExt_{31:12} = 0$ and $ImmExt_{11:0} = Instr_{11:0}$.

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* comes from the register file, and *SrcB* comes from the extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 2-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult*. For an LDR instruction, *ALUControl* should be set to 00 to perform addition. *ALUResult* is sent to the data memory as the address to read, as shown in Figure 7.5.

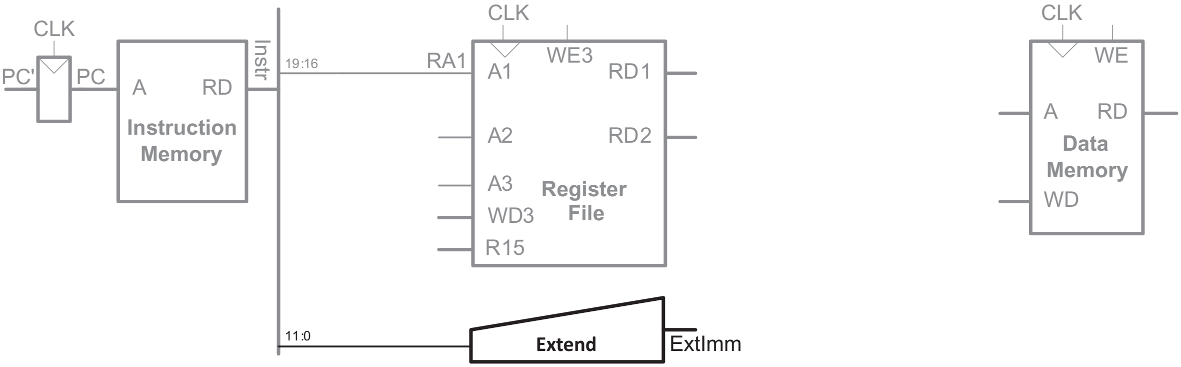

CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
11:0
Extend
ExtImm
CLK
WE
A
RD
Data Memory
WD

Figure 7.4 Zero-extend the immediate

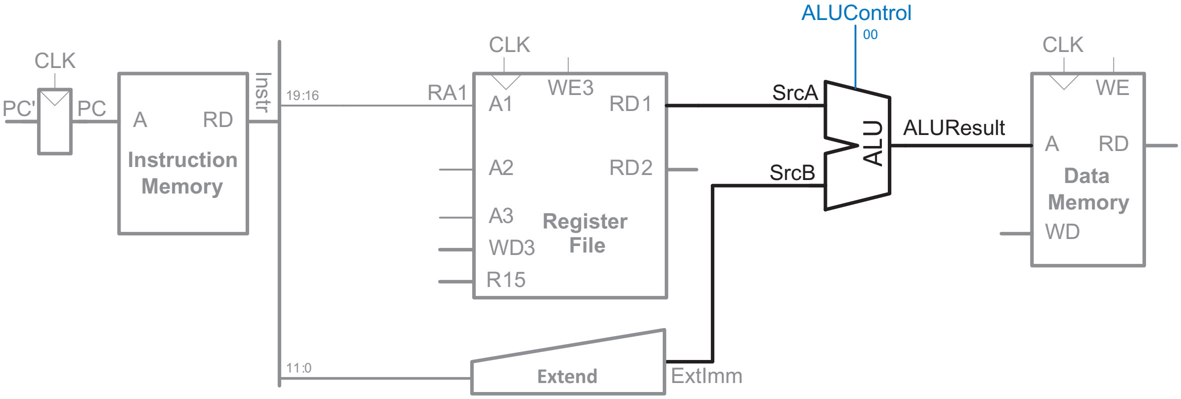

ALUControl
00
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
SrcA
SrcB
ALU
ALUResult
11:0
Extend
ExtImm
CLK
WE
A
RD
Data Memory
WD

Figure 7.5 Compute memory address

The data is read from the data memory onto the *ReadData* bus and then written back to the destination register at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the write port. The destination register for the `LDR` instruction is specified in the *Rd* field, $Instr_{15:12}$, which is connected to the port 3 address input, *A3*, of the register file. The *ReadData* bus is connected to the port 3 write data input, *WD3*, of the register file. A control signal called *RegWrite* is connected to the port 3 write enable input, *WE3*, and is asserted during an `LDR` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

While the instruction is being executed, the processor must compute the address of the next instruction, *PC′*. Because instructions are 32 bits (4 bytes), the next instruction is at $PC+4$. Figure 7.7 uses an adder to

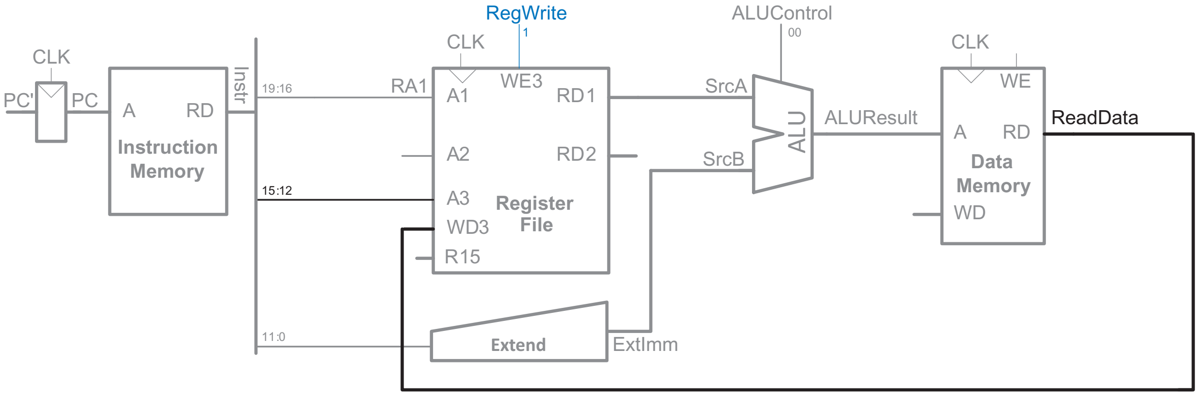

RegWrite
1
ALUControl
00
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
RA1
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
SrcA
SrcB
ALU
ALUResult
WE
A
RD
Data Memory
WD
ReadData
Extend
ExtImm

Figure 7.6 Write data back to register file

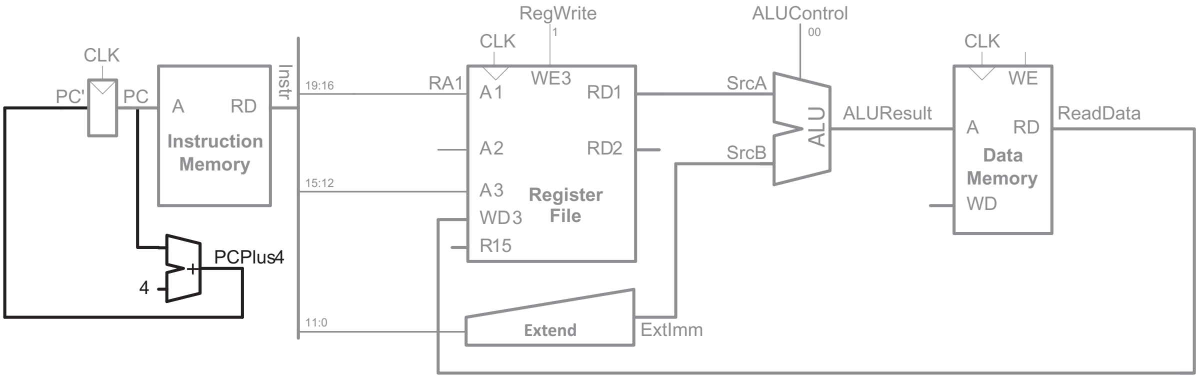

RegWrite
1
ALUControl
00
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
PCPlus4
4
+
RA1
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
SrcA
SrcB
ALU
ALUResult
WE
A
RD
Data Memory
WD
ReadData
Extend
ExtImm

Figure 7.7 Increment program counter

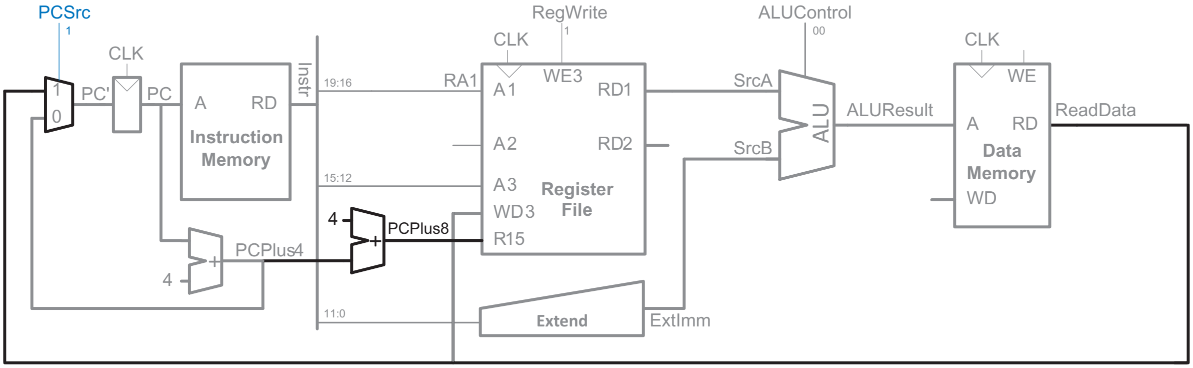

PCSrc
1
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
RA1
4
PCPlus8
PCPlus4
4
RegWrite
1
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
Extend
ExtImm
ALUControl
00
SrcA
SrcB
ALU
ALUResult
CLK
WE
A
RD
Data Memory
WD
ReadData

Figure 7.8 Read or write program counter as R15

increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `LDR` instruction, except for a sneaky case of the base or destination register being R15.

Recall from Section 6.4.6 that in the ARM architecture, reading register R15 returns *PC* + 8. Therefore, another adder is needed to further increment the PC and pass this sum to the *R15* port of the register file. Similarly, writing register R15 updates the PC. Therefore, *PC* may come from the result of the instruction (*ReadData*) rather than *PCPlus4*. A multiplexer chooses between these two possibilities. The *PCSrc* control signal is set to 0 to choose *PCPlus4* or 1 to choose *ReadData*. These PC-related features are highlighted in Figure 7.8.

STR

Next, let us extend the datapath to also handle the `STR` instruction. Like `LDR`, `STR` reads a base address from port 1 of the register file and zero-extends the immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported in the datapath.

The `STR` instruction also reads a second register from the register file and writes it to the data memory. Figure 7.9 shows the new connections for this function. The register is specified in the *Rd* field, $Instr_{15:12}$, which is connected to the *A2* port of the register file. The register value is read onto the *RD2* port. It is connected to the write data (*WD*) port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For an `STR` instruction: *MemWrite* = 1 to write the data to memory; *ALUControl* = 00 to add the base address and offset; and *RegWrite* = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this *ReadData* is ignored because *RegWrite* = 0.

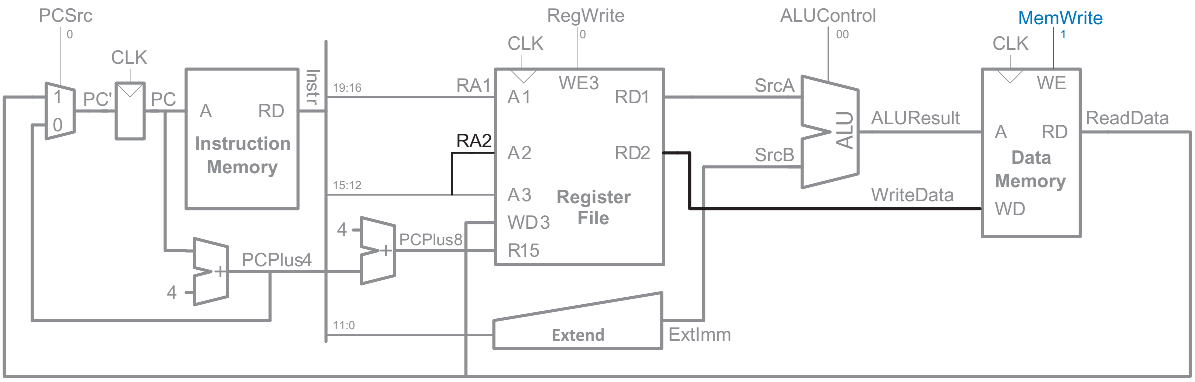

PCSrc
0
CLK
1
0
PC'
PC
A
RD
Instruction
Memory
Instr
19:16
15:12
11:0
4
PCPlus4
4
PCPlus8
RegWrite
0
CLK
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register
File
Extend
ExtImm
ALUControl
00
SrcA
SrcB
ALU
ALUResult
WriteData
MemWrite
1
CLK
WE
A
RD
Data
Memory
WD
ReadData

Figure 7.9 Write data to memory for `STR` instruction

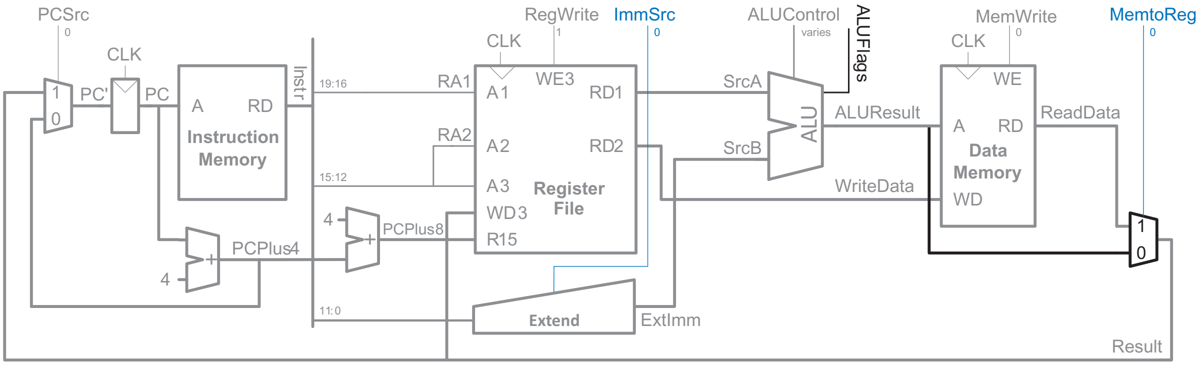

PCSrc
0
CLK
1
0
PC'
PC
A
RD
Instruction
Memory
Instr
19:16
15:12
11:0
4
PCPlus4
4
PCPlus8
RegWrite
1
ImmSrc
0
CLK
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register
File
Extend
ExtImm
ALUControl
varies
ALUFlags
SrcA
SrcB
ALU
ALUResult
WriteData
MemWrite
0
CLK
WE
A
RD
Data
Memory
WD
ReadData
MemtoReg
0
1
0
Result

Figure 7.10 Datapath enhancements for data-processing instructions with immediate addressing

Data-Processing Instructions with Immediate Addressing

Next, consider extending the datapath to handle the data-processing instructions, `ADD`, `SUB`, `AND`, and `ORR`, using the immediate addressing mode. All of these instructions read a source register from the register file and an immediate from the low bits of the instruction, perform some ALU operation on them, and write the result back to a third register. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware using different *ALUControl* signals. As described in Section 5.2.4, *ALUControl* is 00 for `ADD`, 01 for `SUB`, 10 for `AND`, or 11 for `ORR`. The ALU also produces four flags, $ALUFlags_{3:0}$ (Zero, Negative, Carry, oVerflow), that are sent back to the controller.

Figure 7.10 shows the enhanced datapath handling data-processing instructions with an immediate second source. Like `LDR`, the datapath reads the first ALU source from port 1 of the register file and extends the immediate from the low bits of *Instr*. However, data-processing instructions use only an 8-bit immediate rather than a 12-bit immediate. Therefore, we provide the *ImmSrc* control signal to the Extend block. When it is 0, *ExtImm* is zero-extended from $Instr_{7:0}$ for data-processing instructions. When it is 1, *ExtImm* is zero-extended from $Instr_{11:0}$ for `LDR` or `STR`.

For `LDR`, the register file received its write data from the data memory. However, data-processing instructions write *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. The multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0 for data-processing instructions to choose *Result* from *ALUResult*; it is 1 for `LDR` to choose *ReadData*. We do not care about the value of *MemtoReg* for `STR` because `STR` does not write the register file.

Data-Processing Instructions with Register Addressing

Data-processing instructions with register addressing receive their second source from *Rm*, specified by $Instr_{3:0}$, rather than from the immediate. Thus, we must add multiplexers on the inputs of the register file and ALU to select this second source register, as shown in Figure 7.11.

RA2 is chosen from the *Rd* field ($Instr_{15:12}$) for `STR` and the *Rm* field ($Instr_{3:0}$) for data-processing instructions with register addressing based on the *RegSrc* control signal. Similarly, based on the *ALUSrc* control signal, the second source to the ALU is selected from *ExtImm* for instructions using immediates and from the register file for data-processing instructions with register addressing.

B

Finally, we extend the datapath to handle the `B` instruction, as shown in Figure 7.12. The branch instruction adds a 24-bit immediate to $PC + 8$ and writes the result back to the PC. The immediate is multiplied by 4

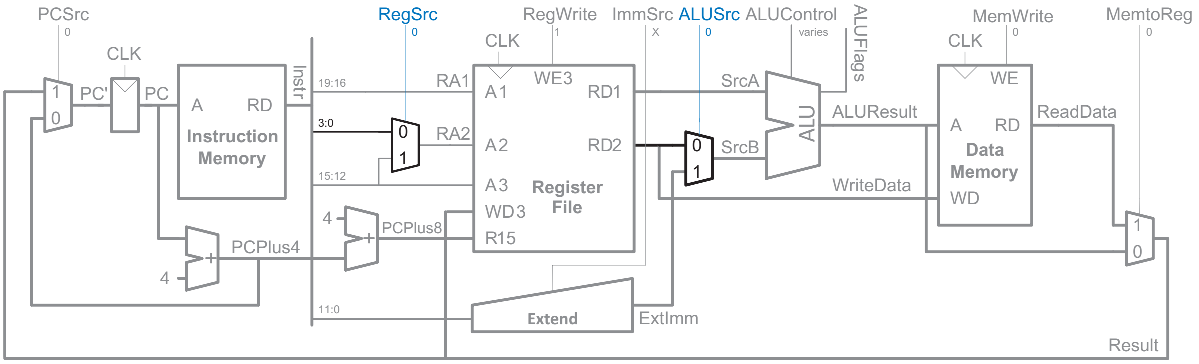

PCSrc
0
CLK
1
0
PC'
PC
A
RD
Instruction Memory
Instr
19:16
3:0
15:12
11:0
RegSrc
0
0
1
RA1
RA2
CLK
RegWrite
1
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
4
PCPlus8
PCPlus4
4
Extend
ExtImm
ImmSrc
X
ALUSrc
0
ALUControl
varies
ALUFlags
SrcA
SrcB
ALU
0
1
ALUResult
WriteData
MemWrite
0
CLK
WE
A
RD
Data Memory
WD
ReadData
MemtoReg
0
1
0
Result

Figure 7.11 Datapath enhancements for data-processing instructions with register addressing

and sign extended. Therefore, the Extend logic needs yet another mode. *ImmSrc* is increased to 2 bits, with the encoding given in Table 7.1.

PC + 8 is read from the first port of the register file. Therefore, a multiplexer is needed to choose R15 as the *RA1* input. This multiplexer is controlled by another bit of *RegSrc*, choosing $Instr_{19:16}$ for most instructions but 15 for `B`.

MemtoReg is set to 0 and *PCSrc* is set to 1 to select the new PC from *ALUResult* for the branch.

This completes the design of the single-cycle processor datapath. We have illustrated not only the design itself but also the design process in which the state elements are identified, and the combinational logic connecting the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

7.3.2 Single-Cycle Control

The control unit computes the control signals based on the *cond*, *op*, and *funct* fields of the instruction ($Instr_{31:28}$, $Instr_{27:26}$, and $Instr_{25:20}$) as well as the flags and whether the destination register is the PC. The controller also stores the current status flags and updates them appropriately. Figure 7.13 shows the entire single-cycle processor with the control unit attached to the datapath.

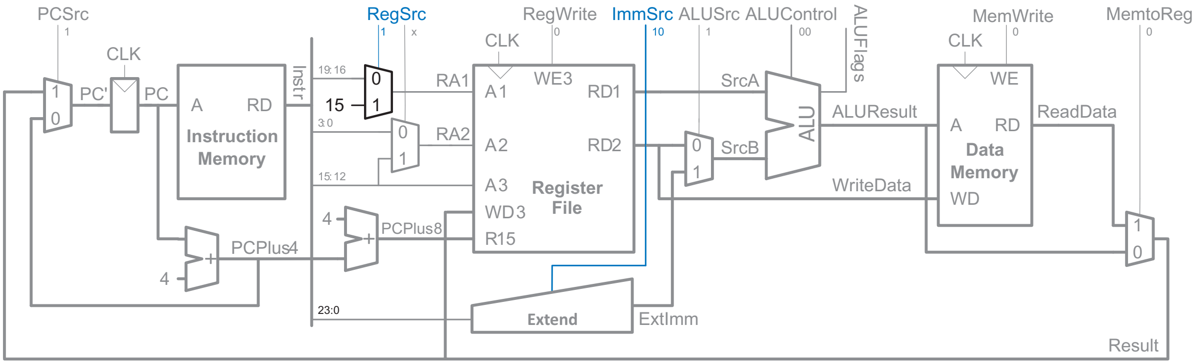

PCSrc
1
CLK
PC'
PC
1
0
A
RD
Instruction Memory
Instr
RegSrc
1
x
19:16
15
0
1
3:0
0
1
15:12
RA1
RA2
4
PCPlus8
PCPlus4
4
23:0
RegWrite
0
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
ImmSrc
10
Extend
ExtImm
ALUSrc
1
SrcA
SrcB
0
1
ALUControl
00
ALU
ALUFlags
ALUResult
WriteData
MemWrite
0
CLK
WE
A
RD
Data Memory
WD
ReadData
MemtoReg
0
1
0
Result

Figure 7.12 Datapath enhancements for `B` instruction

Table 7.1 ImmSrc Encoding

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for `LDR/STR`
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for `B`

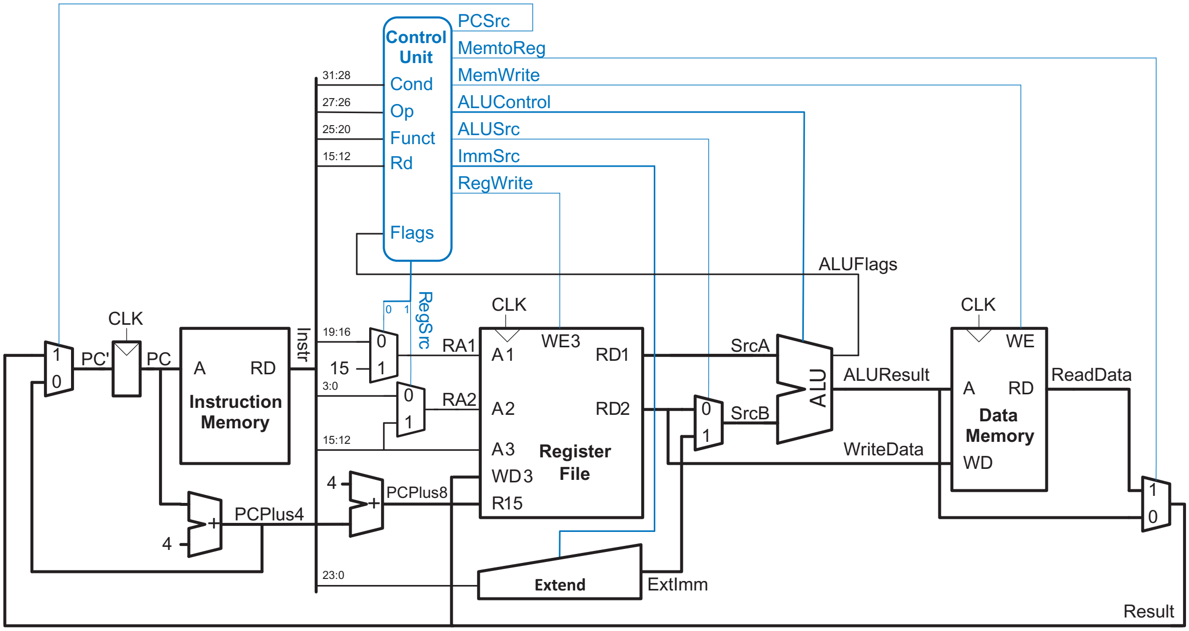
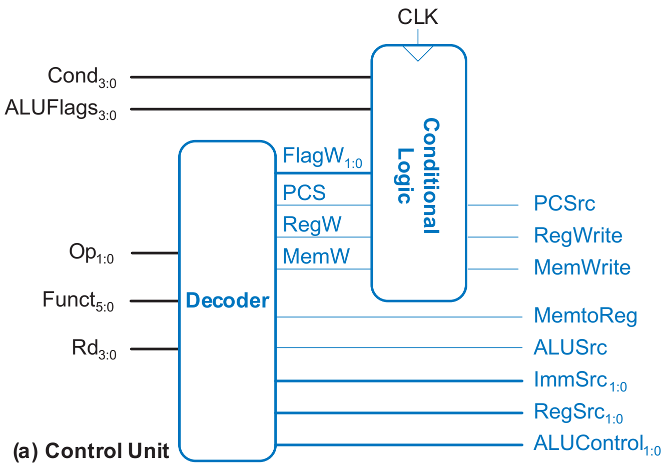

PCSrc
Control Unit
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
31:28
27:26
25:20
15:12
Cond
Op
Funct
Rd
Flags
ALUFlags
RegSrc
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15
3:0
15:12
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
WriteData
WE
A
RD
WD
Data Memory
ReadData
4
PCPlus8
4
PCPlus4
23:0
Extend
ExtImm
Result

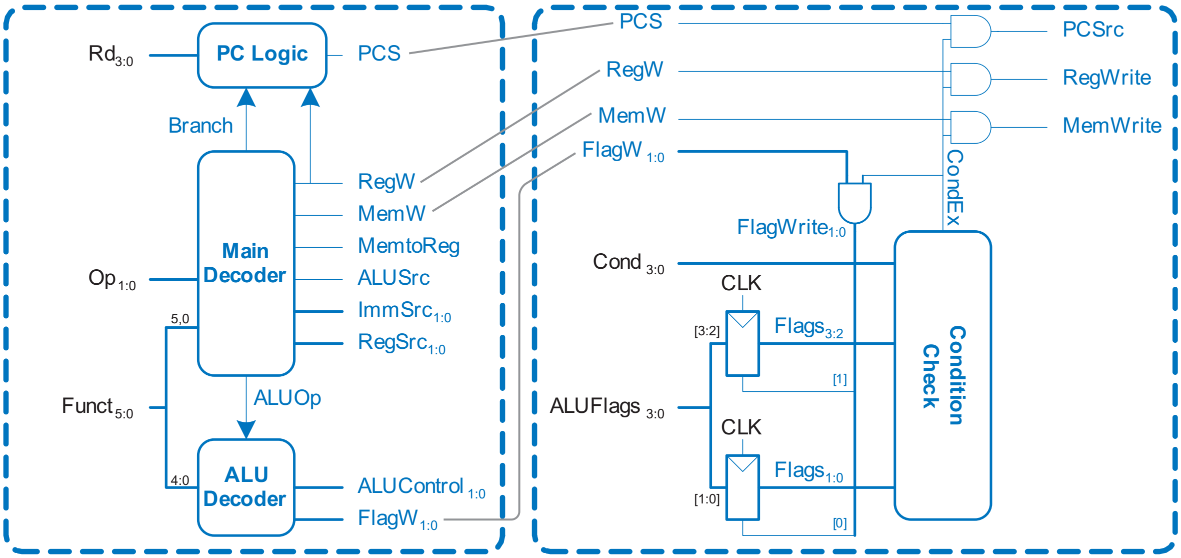
Figure 7.13 Complete single-cycle processor

Figure 7.14 shows a detailed diagram of the controller. We partition the controller into two main parts: the Decoder, which generates control signals based on *Instr*, and the Conditional Logic, which maintains the status flags and only enables updates to architectural state when the instruction should be conditionally executed. The Decoder, shown in Figure 7.14(b), is composed of a Main Decoder that produces most of the control signals, an ALU Decoder that uses the *Funct* field to determine the type of data-processing instruction, and PC Logic to determine whether the PC needs updating due to a branch or a write to R15.

The behavior of the Main Decoder is given by the truth table in Table 7.2. The Main Decoder determines the type of instruction: Data-Processing Register, Data-Processing Immediate, `STR`, `LDR`, or `B`. It produces the appropriate control signals to the datapath. It sends *MemtoReg*, *ALUSrc*, $ImmSrc_{1:0}$, and $RegSrc_{1:0}$ directly to the datapath. However, the write enables *MemW* and *RegW* must pass through the Conditional Logic before becoming datapath signals *MemWrite* and *RegWrite*. These write enables may be killed (reset to 0) by the Conditional Logic if the condition is not satisfied. The Main Decoder also generates the *Branch* and *ALUOp* signals, which are used within the controller to indicate that the instruction is `B` or data-processing, respectively. The logic for the Main Decoder can be developed from the truth table using your favorite techniques for combinational logic design.


CLK
Cond3:0
ALUFlags3:0
Conditional Logic
FlagW1:0
PCS
RegW
MemW
Op1:0
Funct5:0
Rd3:0
Decoder
PCSrc
RegWrite
MemWrite
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
ALUControl1:0

(a) Control Unit


Rd3:0
PC Logic
PCS
Branch
Main Decoder
RegW
MemW
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
Op1:0
5,0
Funct5:0
ALUOp
4:0
ALU Decoder
ALUControl1:0
FlagW1:0
PCS
RegW
MemW
FlagW1:0
PCSrc
RegWrite
MemWrite
CondEx
FlagWrite1:0
Cond3:0
CLK
[3:2]
Flags3:2
[1]
ALUFlags3:0
CLK
Flags1:0
[1:0]
[0]
Condition Check

(b) Decoder **(c) Conditional Logic**

Figure 7.14 Single-cycle control unit

The behavior of the ALU Decoder is given by the truth tables in Table 7.3. For data-processing instructions, the ALU Decoder chooses *ALUControl* based on the type of instruction (ADD, SUB, AND, ORR). Moreover, it asserts *FlagW* to update the status flags when the *S*-bit is set. Note that ADD and SUB update all flags, whereas AND and ORR only update the *N* and *Z* flags, so two bits of *FlagW* are needed: $FlagW_1$ for updating *N* and *Z* ($Flags_{3:2}$), and $FlagW_0$ for updating *C* and *V* ($Flags_{1:0}$). $FlagW_{1:0}$ is killed by the Conditional Logic when the condition is not satisfied ($CondEx = 0$).

Table 7.2 Main Decoder truth table

Op	Funct$_5$	Funct$_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	`STR`	0	X	1	1	01	0	10	0
01	X	1	`LDR`	0	1	0	1	01	1	X0	0
10	X	X	`B`	1	0	0	1	10	0	X1	0

Table 7.3 ALU Decoder truth table

ALUOp	*Funct*$_{4:1}$ (*cmd*)	*Funct*$_0$ (*S*)	Type	*ALUControl*$_{1:0}$	*FlagW*$_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	`ADD`	00 (Add)	00
		1			11
	0010	0	`SUB`	01 (Sub)	00
		1			11
	0000	0	`AND`	10 (And)	00
		1			10
	1100	0	`ORR`	11 (Or)	00
		1			10

The PC Logic checks if the instruction is a write to R15 or a branch such that the PC should be updated. The logic is:

$$PCS = ((Rd == 15)\ \&\ RegW)\ |\ Branch$$

PCS may be killed by the Conditional Logic before it is sent to the datapath as *PCSrc*.

The Conditional Logic, shown in Figure 7.14(c), determines whether the instruction should be executed (*CondEx*) based on the *cond* field and the current values of the *N*, *Z*, *C*, and *V* flags ($Flags_{3:0}$), as was described in Table 6.3. If the instruction should not be executed, the write enables and *PCSrc* are forced to 0 so that the instruction does not change the architectural state. The Conditional Logic also updates some or all of the flags from the *ALUFlags* when *FlagW* is asserted by the ALU Decoder and the instruction's condition is satisfied ($CondEx = 1$).

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an `ORR` instruction with register addressing mode.

Solution: Figure 7.15 illustrates the control signals and flow of data during execution of the `ORR` instruction. The PC points to the memory location holding the instruction, and the instruction memory returns this instruction.

The main flow of data through the register file and ALU is represented with a heavy blue line. The register file reads the two source operands specified by $Instr_{19:16}$ and $Instr_{3:0}$, so *RegSrc* must be 00. *SrcB* should come from the second port of the register file (not *ExtImm*), so *ALUSrc* must be 0. The ALU performs a bitwise OR operation, so *ALUControl* must be 11. The result comes from the ALU, so *MemtoReg* is 0. The result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* = 0.

The updating of *PC* with *PCPlus4* is shown with a heavy gray line. *PCSrc* is 0 to select the incremented *PC*.

Note that data certainly does flow through the nonhighlighted paths, but that the value of that data is unimportant for this instruction. For example, the immediate is extended and data is read from memory, but these values do not influence the next state of the system.

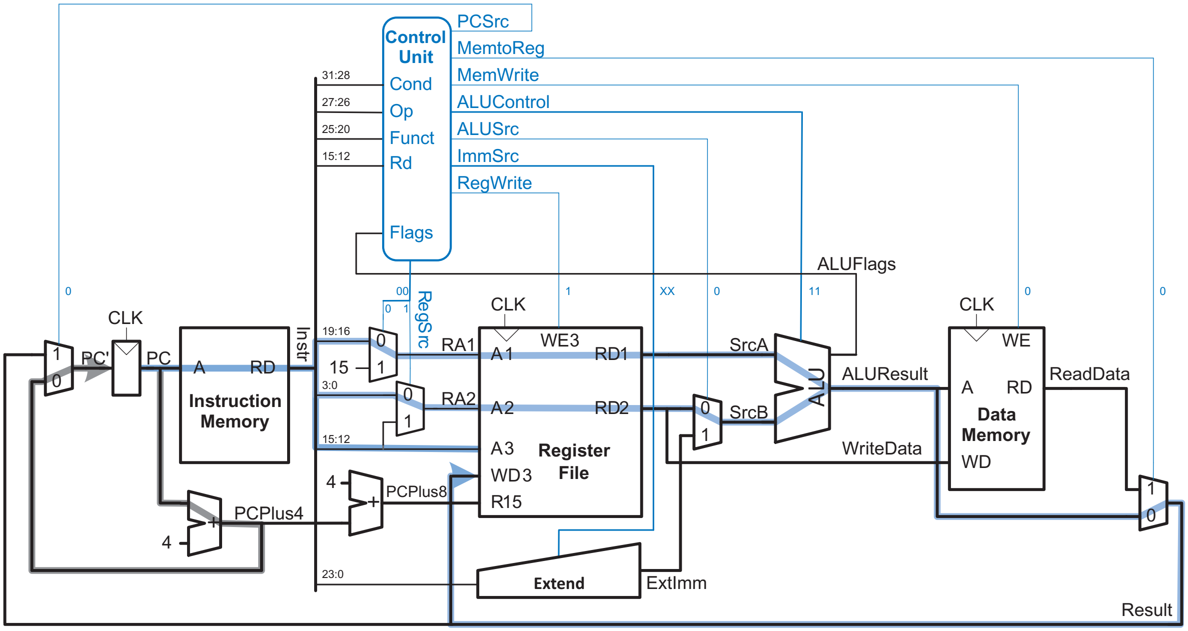

Control Unit
Cond
Op
Funct
Rd
Flags
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
31:28
27:26
25:20
15:12
ALUFlags
0
00
0
1
RegSrc
1
XX
0
11
0
0
CLK
CLK
CLK
1
0
PC'
PC
A
RD
Instruction Memory
Instr
19:16
0
1
15
3:0
0
1
15:12
RA1
RA2
A1
A2
A3
WE3
RD1
RD2
WD3
R15
Register File
SrcA
SrcB
0
1
ALU
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
1
0
4
PCPlus8
4
PCPlus4
23:0
Extend
ExtImm
Result

Figure 7.15 Control signals and data flow while executing an `ORR` instruction

7.3.3 More Instructions

We have considered a limited subset of the full ARM instruction set. In this section, we add support for the compare (`CMP`) instruction and for addressing modes in which the second source is a shifted register. These examples illustrate the principle of how to handle new instructions; with enough effort, you could extend the single-cycle processor to handle every ARM instruction. Moreover, we will see that supporting some instructions simply requires enhancing the decoders, whereas supporting others also requires new hardware in the datapath.

Example 7.2 `CMP` INSTRUCTION

The compare instruction, `CMP`, subtracts *SrcB* from *SrcA* and sets the flags but does not write the difference to a register. The datapath is already capable of this task. Determine the necessary changes to the controller to support `CMP`.

Solution: Introduce a new control signal called *NoWrite* to prevent writing *Rd* during `CMP`. (This signal would also be helpful for other instructions such as `TST` that do not write a register.) We extend the ALU Decoder to produce this signal and the *RegWrite* logic to accept it, as highlighted in blue in Figure 7.16. The enhanced ALU Decoder truth table is given in Table 7.4, with the new instruction and signal also highlighted.

Example 7.3 ENHANCED ADDRESSING MODE: REGISTERS WITH CONSTANT SHIFTS

So far, we assumed that data-processing instructions with register addressing did not shift the second source register. Enhance the single-cycle processor to support a shift by an immediate.

Solution: Insert a shifter before the ALU. Figure 7.17 shows the enhanced datapath. The shifter uses $Instr_{11:7}$ to specify the shift amount and $Instr_{6:5}$ to specify the shift type.

7.3.4 Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical paths for the `LDR` instruction are shown in Figure 7.18 with a heavy blue line. It starts with the PC loading a new address on the rising edge of the clock. The instruction memory reads the new instruction. The Main Decoder computes $RegSrc_0$, which drives the multiplexer to choose $Instr_{19:16}$ as *RA1*, and the register file reads this register as *SrcA*. While the register file is reading, the immediate field is zero-extended and selected at the *ALUSrc* multiplexer to determine *SrcB*. The ALU adds *SrcA* and *SrcB* to find the effective address. The data memory reads from this address. The *MemtoReg* multiplexer selects *ReadData*.

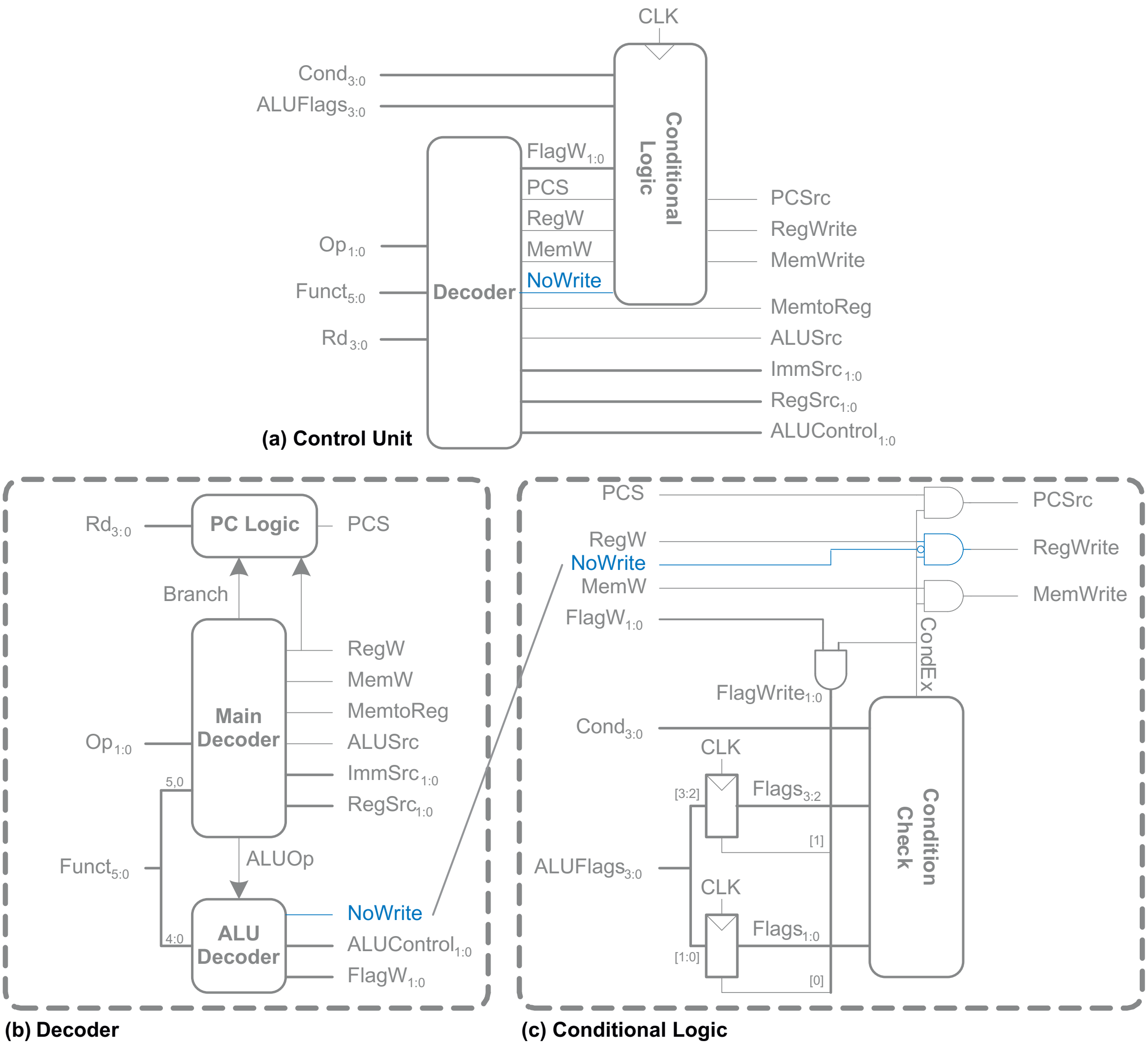

CLK
Cond3:0
ALUFlags3:0
Conditional Logic
FlagW1:0
PCS
RegW
MemW
NoWrite
Op1:0
Funct5:0
Rd3:0
Decoder
PCSrc
RegWrite
MemWrite
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
ALUControl1:0
(a) Control Unit
Rd3:0
PC Logic
PCS
Branch
Main Decoder
RegW
MemW
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
Op1:0
Funct5:0
5,0
4:0
ALUOp
ALU Decoder
NoWrite
ALUControl1:0
FlagW1:0
(b) Decoder
PCS
RegW
NoWrite
MemW
FlagW1:0
PCSrc
RegWrite
MemWrite
CondEx
FlagWrite1:0
Cond3:0
CLK
[3:2]
Flags3:2
[1]
ALUFlags3:0
CLK
Flags1:0
[1:0]
[0]
Condition Check
(c) Conditional Logic

Figure 7.16 Controller modification for `CMP`

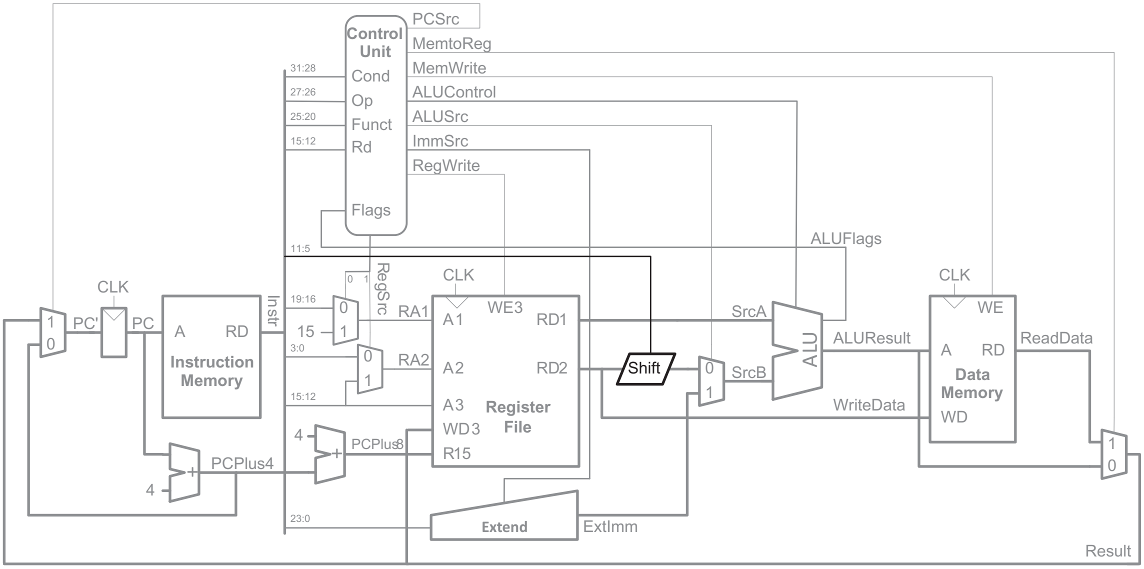
Finally, *Result* must set up at the register file before the next rising clock edge so that it can be properly written. Hence, the cycle time is:

$$T_{c1} = t_{pcq\_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \tag{7.2}$$

We use the subscript 1 to distinguish this cycle time from that of subsequent processor designs. In most implementation technologies, the

Table 7.4 ALU Decoder truth table enhanced for `CMP`

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	$NoWrite$
0	X	X	Not DP	00	00	0
1	0100	0	`ADD`	00	00	0
		1			11	0
	0010	0	`SUB`	01	00	0
		1			11	0
	0000	0	`AND`	10	00	0
		1			10	0
	1100	0	`ORR`	11	00	0
		1			10	0
	1010	1	`CMP`	01	11	1



PCSrc
Control Unit
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
31:28
27:26
25:20
15:12
Cond
Op
Funct
Rd
Flags
11:5
ALUFlags
CLK
CLK
CLK
RegSrc
0 1
19:16
15
3:0
15:12
Instr
PC'
PC
A RD
Instruction Memory
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
Shift
SrcA
SrcB
ALU
ALUResult
WriteData
WE
A RD
Data Memory
WD
ReadData
4
PCPlus4
4
PCPlus8
23:0
Extend
ExtImm
Result
1
0

Figure 7.17 Enhanced datapath for register addressing with constant shifts

ALU, memory, and register file are substantially slower than other combinational blocks. Therefore, the cycle time simplifies to:

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup} \tag{7.3}$$

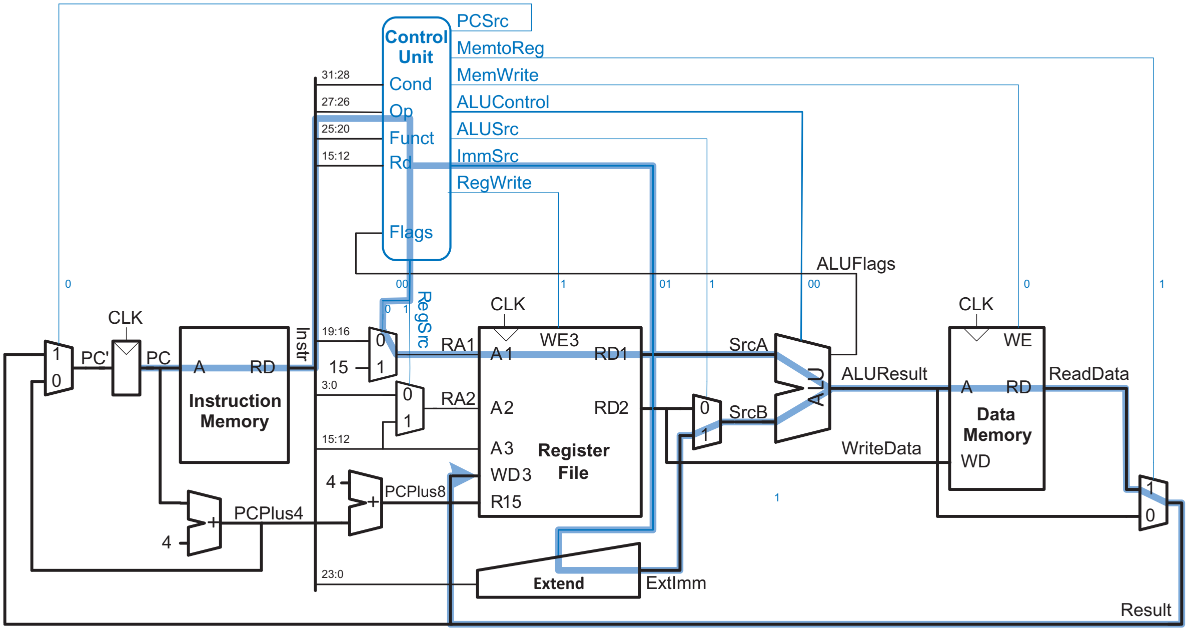

Control Unit
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
Cond
Op
Funct
Rd
Flags
31:28
27:26
25:20
15:12
ALUFlags
CLK
PC'
PC
A
RD
Instruction Memory
Instr
RegSrc
19:16
15
3:0
15:12
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
WriteData
WE
Data Memory
WD
ReadData
Result
PCPlus4
PCPlus8
4
Extend
ExtImm
23:0
0
1
00
01

Figure 7.18 `LDR` **critical path**

The numerical values of these times will depend on the specific implementation technology.

Other instructions have shorter critical paths. For example, data-processing instructions do not need to access data memory. However, we are disciplining ourselves to synchronous sequential design, so the clock period is constant and must be long enough to accommodate the slowest instruction.

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle processor in a 16-nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.5. Help him compute the execution time for a program with 100 billion instructions.

Solution: According to Equation 7.3, the cycle time of the single-cycle processor is T_{c1} = 40 + 2(200) + 70 + 100 + 120 + 2(25) + 60 = 840 ps. According to Equation 7.1, the total execution time is $T_1 = (100 \times 10^9$ instruction) (1 cycle/instruction) (840×10^{-12} s/cycle) = 84 seconds.

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three notable weaknesses. First, it requires separate memories for instructions and data, whereas most processors only have a single external memory holding both instructions and data. Second, it requires a clock cycle long enough to support the slowest instruction (`LDR`), even though most instructions could be faster. Finally, it requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast.

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps. In each short step, the processor can read or write the memory or register file or use the ALU. The instruction is read in one step and data can be read or written in a later step, so the processor can use a single memory for both. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones. And the processor needs only one adder, which is reused for different purposes on different steps.

We design a multicycle processor following the same procedure we used for the single-cycle processor. First, we construct a datapath by connecting the architectural state elements and memories with combinational logic. But, this time, we also add nonarchitectural state elements to hold intermediate results between the steps. Then, we design the controller. The controller produces different signals on different steps during execution of a single instruction, so now it is a finite state machine rather than combinational logic. Finally, we analyze the performance of the multicycle processor and compare it with the single-cycle processor.

7.4.1 Multicycle Datapath

Again, we begin our design with the memory and architectural state of the processor, as shown in Figure 7.19. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic, and it is feasible because we can read the instruction in one cycle, then read or write the data in a separate cycle. The PC and register file remain unchanged. As with the single-cycle processor, we gradually build the datapath by adding components to handle each step of each instruction.

The PC contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.20 shows that the PC is simply connected to the address input of the memory. The instruction is read and stored in a new nonarchitectural instruction register (IR) so that it is available for future cycles. The IR receives an enable signal, called *IRWrite*, which is asserted when the IR should be loaded with a new instruction.

LDR

As we did with the single-cycle processor, we first work out the datapath connections for the LDR instruction. After fetching LDR, the next step is

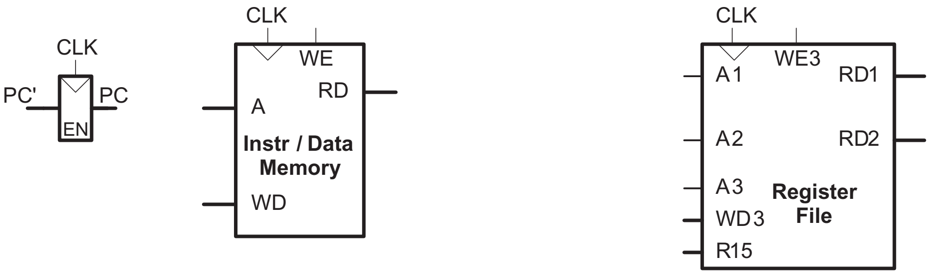

CLK
PC'
PC
EN
CLK
WE
RD
A
Instr / Data
Memory
WD
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register
File

Figure 7.19 State elements with unified instruction/data memory

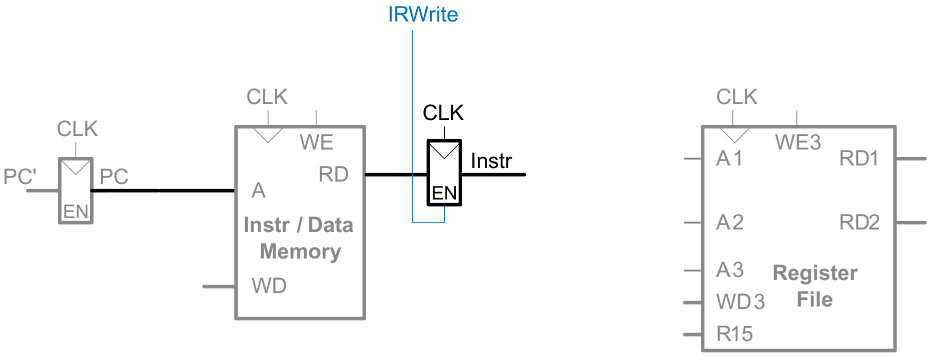

IRWrite
CLK
PC'
PC
EN
CLK
WE
RD
A
Instr / Data
Memory
WD
CLK
EN
Instr
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register
File

Figure 7.20 Fetch instruction from memory

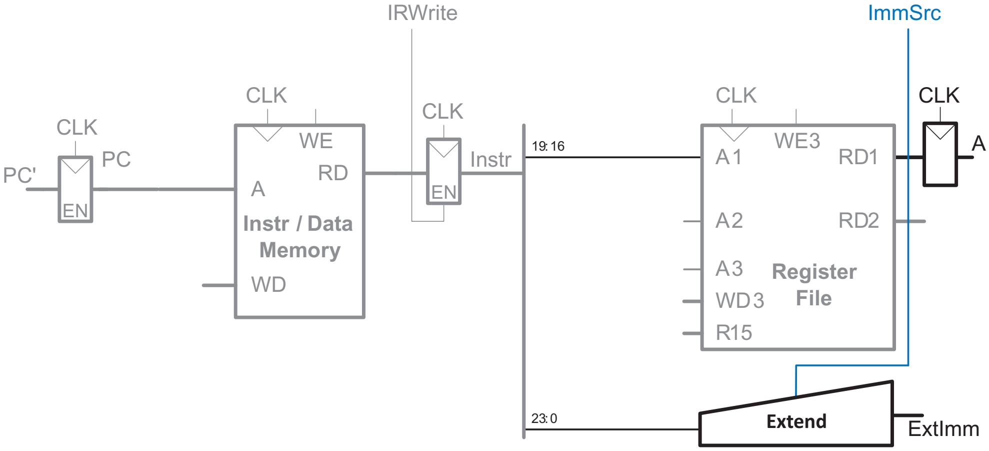

IRWrite
ImmSrc
CLK
PC'
PC
EN
WE
RD
A
Instr / Data Memory
WD
Instr
19:16
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
A
Extend
ExtImm

Figure 7.21 Read one source from register file and extend the second source from the immediate field

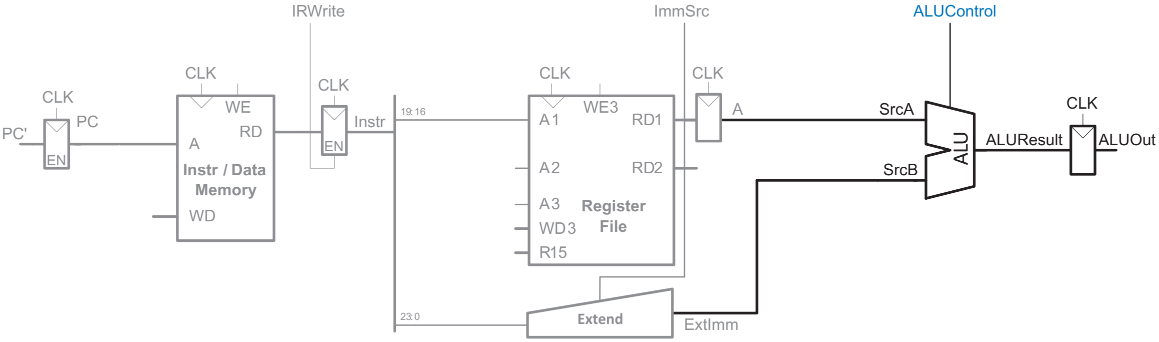

IRWrite
ImmSrc
ALUControl
CLK
PC'
PC
EN
WE
RD
A
Instr / Data Memory
WD
Instr
19:16
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
A
SrcA
SrcB
ALU
ALUResult
ALUOut
Extend
ExtImm

Figure 7.22 Add base address to offset

to read the source register containing the base address. This register is specified in the *Rn* field, $Instr_{19:16}$. These bits of the instruction are connected to address input *A1* of the register file, as shown in Figure 7.21. The register file reads the register into *RD1*. This value is stored in another nonarchitectural register, *A*.

The `LDR` instruction also requires a 12-bit offset, found in the immediate field of the instruction, $Instr_{11:0}$, which must be zero-extended to 32 bits, as shown in Figure 7.21. As in the single-cycle processor, the Extend block takes an *ImmSrc* control signal to specify an 8-, 12-, or 24-bit immediate to extend for various types of instructions. The 32-bit extended immediate is called *ExtImm*. To be consistent, we might store *ExtImm* in another nonarchitectural register. However, *ExtImm* is a combinational function of *Instr* and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load is the sum of the base address and offset. We use an ALU to compute this sum, as shown in Figure 7.22. *ALUControl* should be set to 00 to perform the addition. *ALUResult* is stored in a nonarchitectural register called *ALUOut*.

The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the memory address, *Adr*, from either the PC or *ALUOut* based on the *AdrSrc* select, as shown in Figure 7.23. The data read from memory is stored in another nonarchitectural register, called *Data*. Note that the address multiplexer permits us to reuse the memory during the `LDR` instruction. On a first step, the address is taken from the PC to fetch the instruction. On a later step, the address is taken from *ALUOut* to load the data. Hence, *AdrSrc* must have different values on different steps. In Section 7.4.2, we develop the FSM controller that generates these sequences of control signals.

Finally, the data is written back to the register file, as shown in Figure 7.24. The destination register is specified by the *Rd* field of the instruction, $Instr_{15:12}$. The result comes from the *Data* register. Instead of

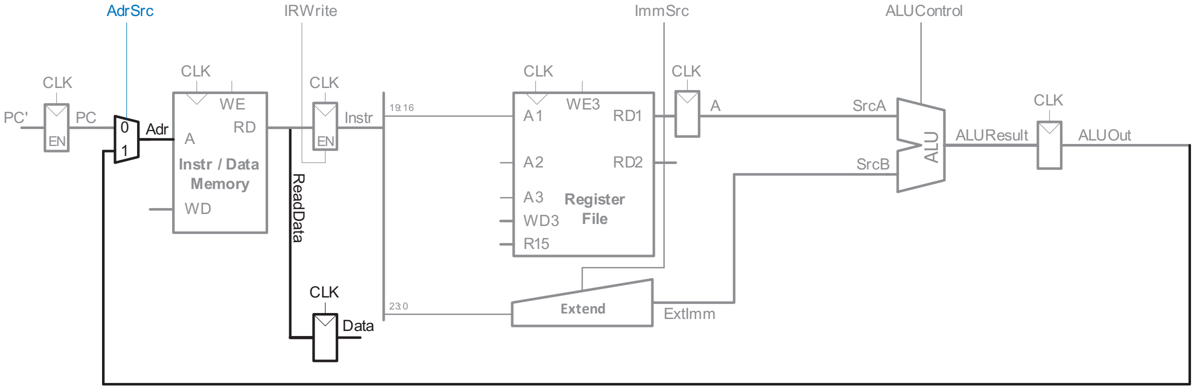

AdrSrc
IRWrite
ImmSrc
ALUControl
CLK
PC'
EN
PC
0
1
Adr
WE
A
RD
Instr / Data
Memory
WD
Instr
ReadData
19:16
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register
File
A
SrcA
SrcB
ALU
ALUResult
ALUOut
Extend
ExtImm
Data

Figure 7.23 Load data from memory

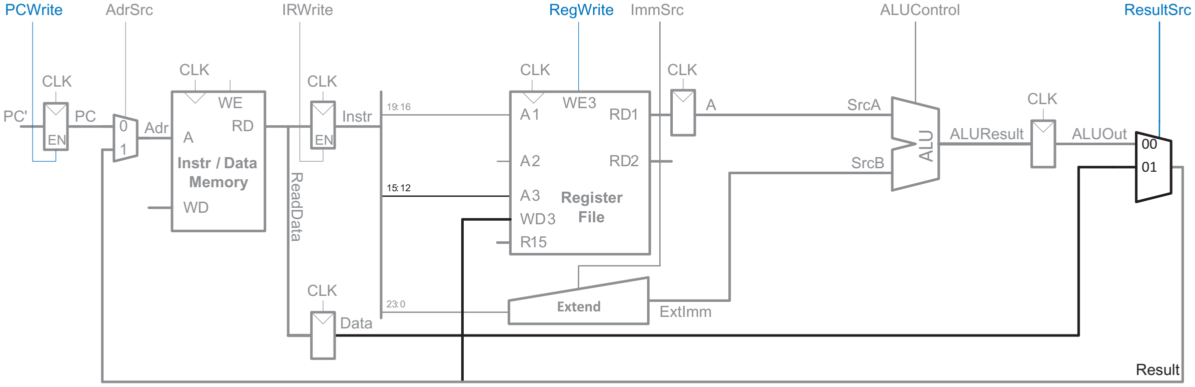

PCWrite
AdrSrc
IRWrite
RegWrite
ImmSrc
ALUControl
ResultSrc
CLK
PC'
EN
PC
0
1
Adr
WE
A
RD
Instr / Data
Memory
WD
Instr
ReadData
19:16
15:12
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register
File
A
SrcA
SrcB
ALU
ALUResult
ALUOut
00
01
Extend
ExtImm
Data
Result

Figure 7.24 Write data back to register file

connecting the *Data* register directly to the register file *WD3* write port, let us add a multiplexer on the *Result* bus to choose either *ALUOut* or *Data* before feeding *Result* back to the register file write port. This will be helpful because other instructions will need to write a result from the ALU. The *RegWrite* signal is 1 to indicate that the register file should be updated.

While all this is happening, the processor must update the program counter by adding 4 to the old *PC*. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU during the fetch step because it is not busy. To do so, we must insert source multiplexers to choose *PC* and the constant 4 as ALU inputs, as shown in Figure 7.25. A multiplexer controlled by *ALUSrcA* chooses either *PC* or register *A* as *SrcA*. Another multiplexer chooses either 4 or *ExtImm* as *SrcB*. To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter. The *ResultSrc* multiplexer chooses this sum from *ALUResult* rather than *ALUOut*; this requires a third input. The *PCWrite* control signal enables the PC to be written only on certain cycles.

Again, we face the ARM architecture idiosyncrasy that reading R15 returns *PC* + 8 and writing R15 updates the PC. First, consider R15 reads. We already computed *PC* + 4 during the fetch step, and the sum is available in the PC register. Thus, during the second step, we obtain *PC* + 8 by adding four to the updated PC using the ALU. *ALUResult* is selected as the *Result* and fed to the R15 input port of the register file. Figure 7.26 shows the completed `LDR` datapath with this new connection. Thus, a read of R15, which also occurs during the second step, produces the value *PC* + 8 on the read data output of the register file. Writes to R15 require writing the PC register instead of the register file. Thus, in the final step of the instruction, *Result* must be routed to the PC register (instead of to the register file) and *PCWrite* must be asserted (instead of *RegWrite*). The datapath already accommodates this, so no datapath changes are required.

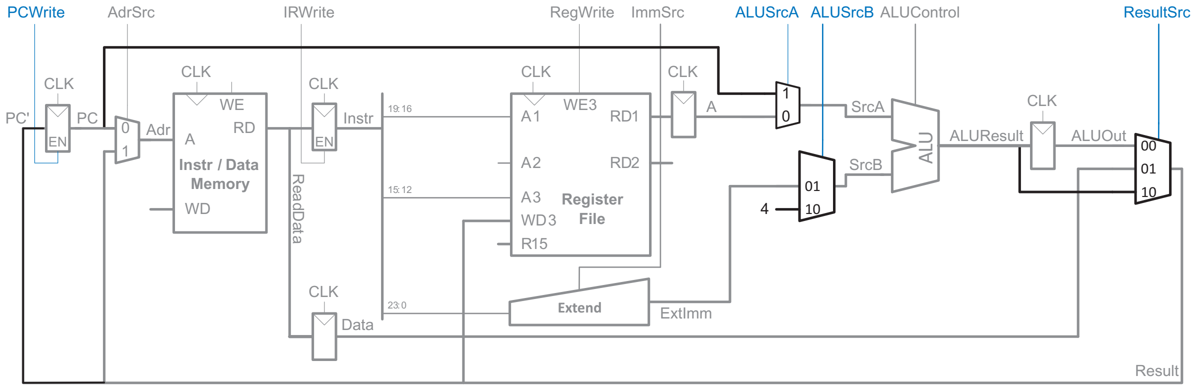

PCWrite
AdrSrc
IRWrite
RegWrite
ImmSrc
ALUSrcA
ALUSrcB
ALUControl
ResultSrc
CLK
PC'
EN
PC
0
1
Adr
CLK
WE
A
RD
Instr / Data
Memory
WD
ReadData
CLK
Instr
EN
19:16
15:12
23:0
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register
File
CLK
A
1
0
SrcA
SrcB
01
10
4
ALU
ALUResult
CLK
ALUOut
00
01
10
Extend
ExtImm
CLK
Data
Result

Figure 7.25 **Increment PC by 4**

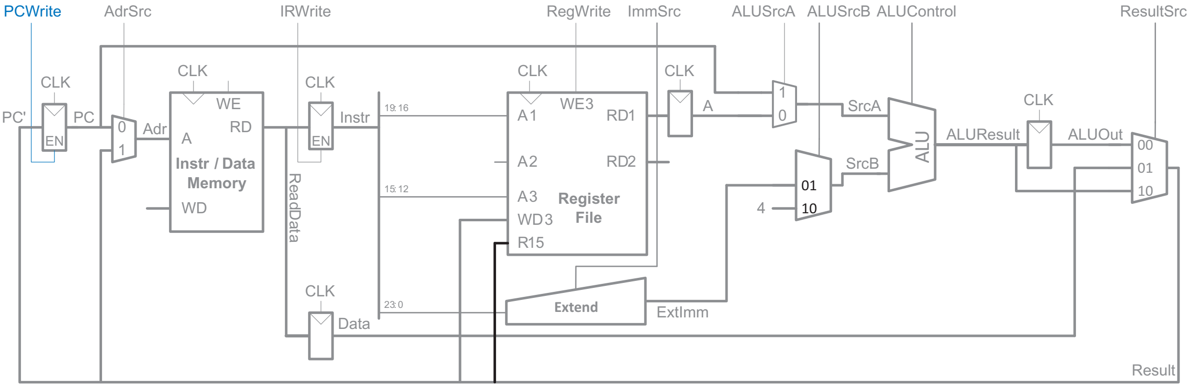

PCWrite
AdrSrc
IRWrite
RegWrite
ImmSrc
ALUSrcA
ALUSrcB
ALUControl
ResultSrc
CLK
PC'
PC
EN
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
ReadData
19:16
15:12
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
Extend
ExtImm
Data
A
SrcA
SrcB
4
ALU
ALUResult
ALUOut
Result

Figure 7.26 Handle R15 reads and writes

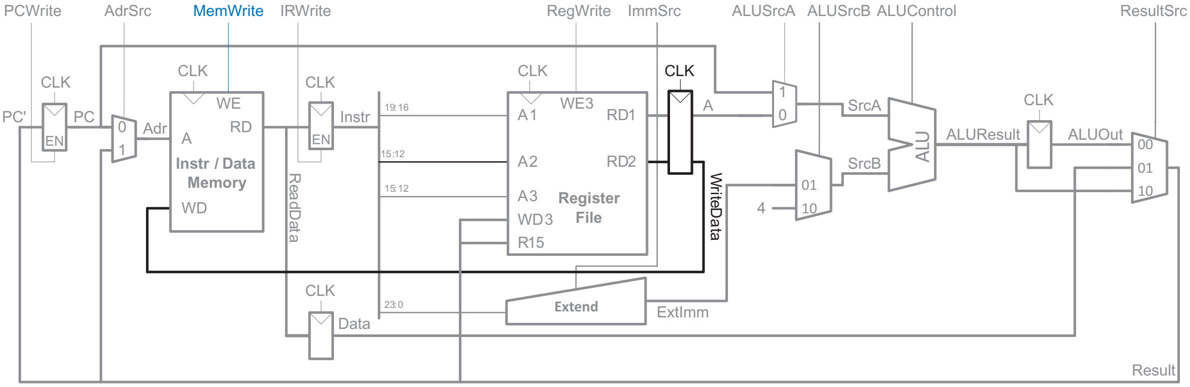

PCWrite
AdrSrc
MemWrite
IRWrite
RegWrite
ImmSrc
ALUSrcA
ALUSrcB
ALUControl
ResultSrc
CLK
PC'
PC
EN
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
ReadData
19:16
15:12
23:0
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
WriteData
Extend
ExtImm
Data
A
SrcA
SrcB
4
ALU
ALUResult
ALUOut
Result

Figure 7.27 Enhanced datapath for `STR` instruction

STR

Next, let us extend the datapath to handle the `STR` instruction. Like `LDR`, `STR` reads a base address from port 1 of the register file and extends the immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by existing hardware in the datapath.

The only new feature of `STR` is that we must read a second register from the register file and write it into the memory, as shown in Figure 7.27. The register is specified in the *Rd* field of the instruction, $Instr_{15:12}$, which is connected to the second port of the register file. When the register is read, it is stored in a nonarchitectural register, *WriteData*. On the next step, it is sent to the write data port (*WD*) of the data memory to be written.

The memory receives the *MemWrite* control signal to indicate that the write should occur.

Data-Processing Instructions with Immediate Addressing

Data-processing instructions with immediate addressing read the first source from *Rn* and extend the second source from an 8-bit immediate. They operate on these two sources and then write the result back to the register file. The datapath already contains all the connections necessary for these steps. The ALU uses the *ALUControl* signal to determine the type of data-processing instruction to execute. The *ALUFlags* are sent back to the controller to update the *Status* register.

Data-Processing Instructions with Register Addressing

Data-processing instructions with register addressing select the second source from the register file. The register is specified in the *Rm* field, $Instr_{3:0}$, so we insert a multiplexer to choose this field as *RA2* for the register file. We also extend the *SrcB* multiplexer to accept the value read from the register file, as shown in Figure 7.28. Otherwise, the behavior is the same as for data-processing instructions with immediate addressing.

B

The branch instruction `B` reads $PC+8$ and a 24-bit immediate, sums them, and adds the result to the PC. Recall from Section 6.4.6 that a read to R15 returns $PC+8$, so we add a multiplexer to choose R15 as *RA1* for the register file, as shown in Figure 7.29. The rest of the hardware to perform the addition and write the PC is already present in the datapath.

This completes the design of the multicycle datapath. The design process is much like that of the single-cycle processor in that hardware is systematically connected between the state elements to handle each instruction. The main difference is that the instruction is executed in several steps. Nonarchitectural registers are inserted to hold the results

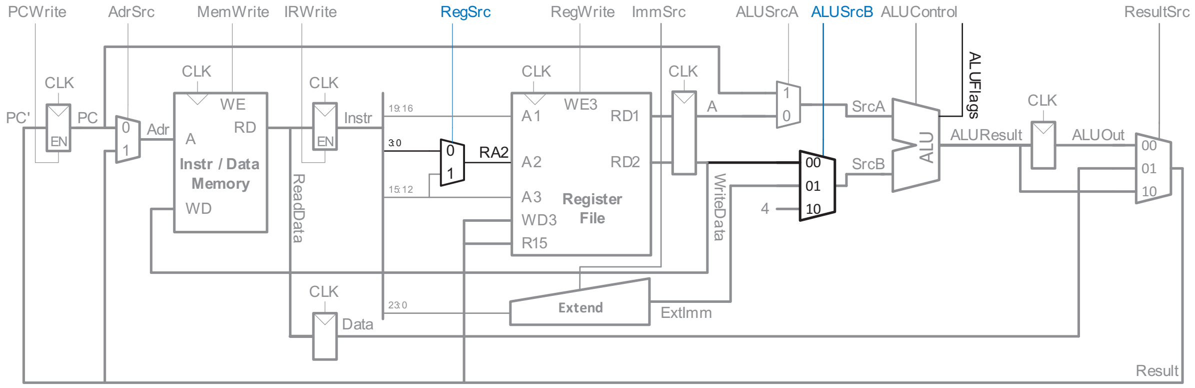

PCWrite
AdrSrc
MemWrite
IRWrite
RegSrc
RegWrite
ImmSrc
ALUSrcA
ALUSrcB
ALUControl
ResultSrc
CLK
PC'
EN
PC
0
1
Adr
CLK
WE
RD
A
Instr / Data
Memory
WD
ReadData
CLK
EN
Instr
19:16
3:0
15:12
0
1
RA2
CLK
WE3
A1
RD1
A2
RD2
A3
Register
File
WD3
R15
CLK
A
1
0
WriteData
00
01
10
4
SrcA
SrcB
ALU
ALUFlags
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
23:0
Extend
ExtImm
Result

Figure 7.28 Enhanced datapath for data-processing instructions with register addressing

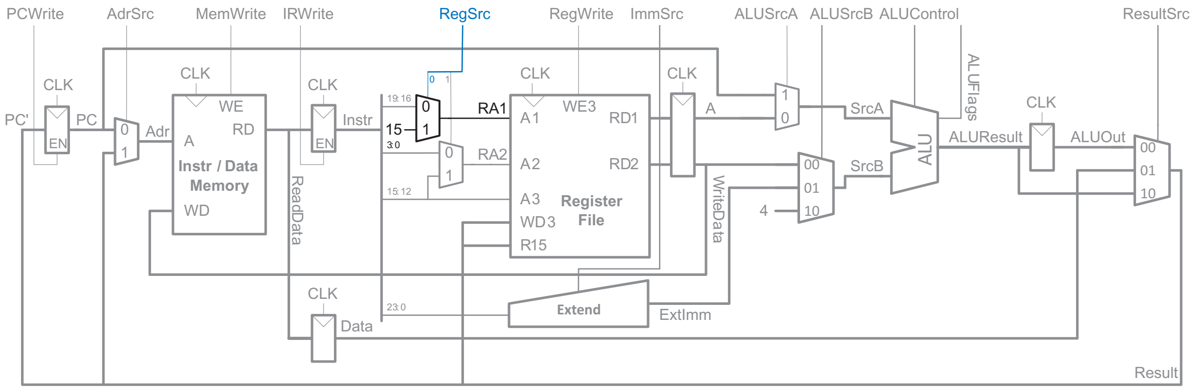

PCWrite
AdrSrc
MemWrite
IRWrite
RegSrc
RegWrite
ImmSrc
ALUSrcA
ALUSrcB
ALUControl
ResultSrc
CLK
PC'
PC
EN
0
1
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
ReadData
19:16
15
3:0
15:12
23:0
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
Data
Extend
ExtImm
WriteData
SrcA
SrcB
00
01
10
4
ALU
ALUFlags
ALUResult
ALUOut
Result

Figure 7.29 **Enhanced datapath for the** `B` **instruction**

of each step. In this way, the memory can be shared for instructions and data and the ALU can be reused several times, reducing hardware costs. In the next section, we develop an FSM controller to deliver the appropriate sequence of control signals to the datapath on each step of each instruction.

7.4.2 Multicycle Control

As in the single-cycle processor, the control unit computes the control signals based on the *cond*, *op*, and *funct* fields of the instruction ($Instr_{31:28}$, $Instr_{27:26}$, and $Instr_{25:20}$) as well as the flags and whether the destination register is the PC. The controller also stores the current status flags and updates them appropriately. Figure 7.30 shows the entire multicycle processor with the control unit attached to the datapath. The datapath is shown in black and the control unit is shown in blue.

As in the single-cycle processor, the control unit is partitioned into Decoder and Conditional Logic blocks, as shown in Figure 7.31(a). The Decoder is decomposed further in Figure 7.31(b). The combinational Main Decoder of the single-cycle processor is replaced with a Main FSM in the multicycle processor to produce a sequence of control signals on the appropriate cycles. We design the Main FSM as a Moore machine so that the outputs are only a function of the current state. However, we will see during the state machine design that *ImmSrc* and *RegSrc* are a function of *Op* rather than the current state, so we also use a small Instruction Decoder to compute these signals, as will be described in Table 7.6. The ALU Decoder and PC Logic are identical to those in the single-cycle processor. The Conditional Logic is almost identical to that of the single-cycle processor. We add a *NextPC* signal to force a write to the PC when we compute $PC + 4$. We also delay *CondEx* by one cycle

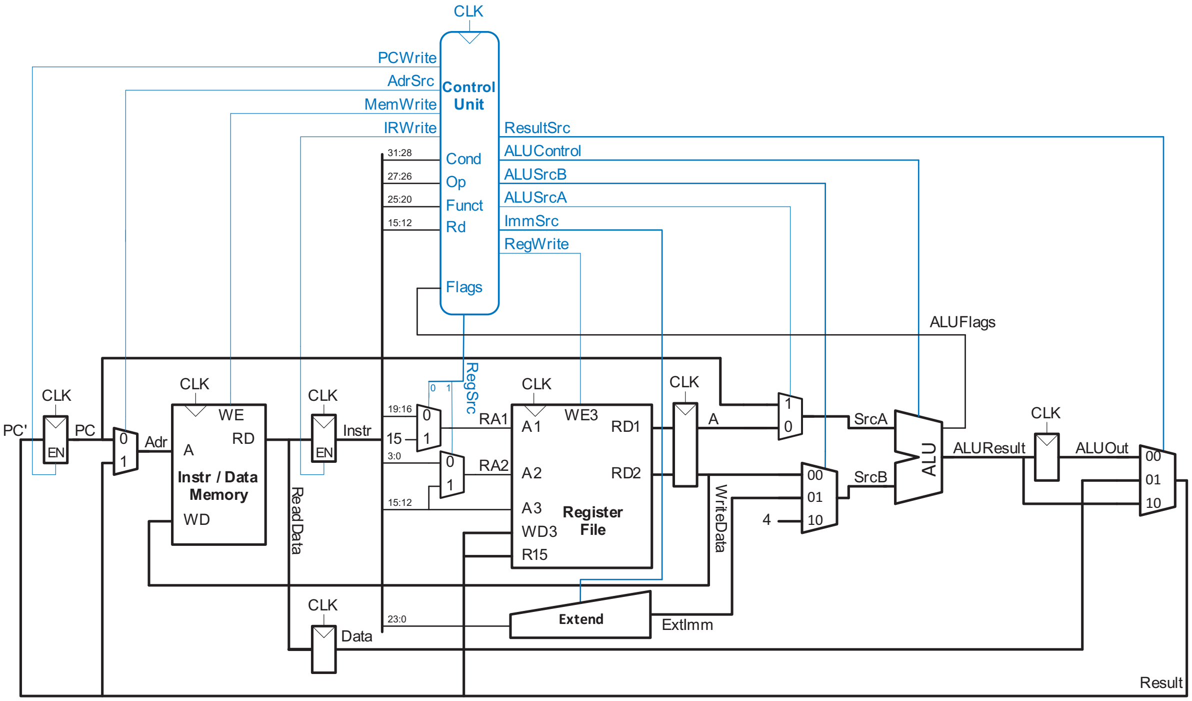

CLK
PCWrite
AdrSrc
MemWrite
IRWrite
Control Unit
ResultSrc
ALUControl
ALUSrcB
ALUSrcA
ImmSrc
RegWrite
31:28
27:26
25:20
15:12
Cond
Op
Funct
Rd
Flags
ALUFlags
RegSrc
PC'
CLK
EN
PC
0
1
Adr
CLK
WE
A
RD
Instr / Data Memory
WD
CLK
EN
Instr
ReadData
19:16
15
3:0
15:12
0
1
0
1
RA1
RA2
CLK
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register File
CLK
A
WriteData
1
0
SrcA
00
01
10
4
SrcB
ALU
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
23:0
Extend
ExtImm
Result

Figure 7.30 Complete multicycle processor

before sending it to *PCWrite*, *RegWrite*, and *MemWrite* so that updated condition flags are not seen until the end of an instruction. The remainder of this section develops the state transition diagram for the Main FSM.

The Main FSM produces multiplexer select, register enable, and memory write enable signals for the datapath. To keep the following state transition diagrams readable, only the relevant control signals are listed. Select signals are listed only when their value matters; otherwise, they are don't care. Enable signals (*RegW*, *MemW*, *IRWrite*, and *NextPC*) are listed only when they are asserted; otherwise, they are 0.

The first step for any instruction is to fetch the instruction from memory at the address held in the PC and to increment the PC to the next instruction. The FSM enters this Fetch state on reset. The control signals are shown in Figure 7.32. The data flow on this step is shown in Figure 7.33, with the instruction fetch highlighted in blue and the PC increment highlighted in gray. To read memory, *AdrSrc* = 0, so the address is taken from the PC. *IRWrite* is asserted to write the instruction into the instruction register, *IR*. Meanwhile, the PC should be incremented by 4 to point to the next instruction. Because the ALU is not being used for anything else, the processor can

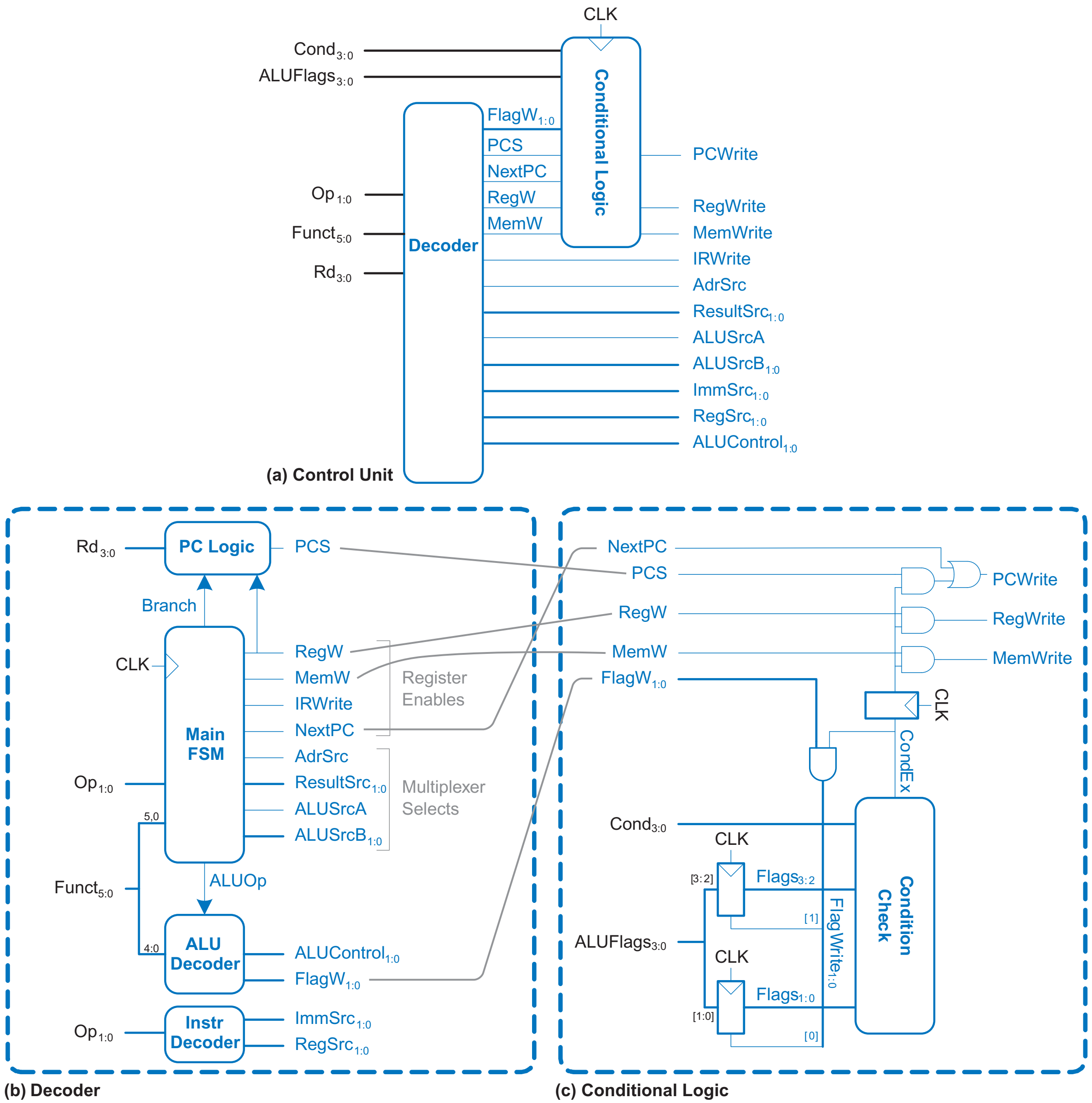

CLK
Cond$_{3:0}$
ALUFlags$_{3:0}$
Conditional Logic
FlagW$_{1:0}$
PCS
NextPC
RegW
MemW
Op$_{1:0}$
Funct$_{5:0}$
Rd$_{3:0}$
Decoder
PCWrite
RegWrite
MemWrite
IRWrite
AdrSrc
ResultSrc$_{1:0}$
ALUSrcA
ALUSrcB$_{1:0}$
ImmSrc$_{1:0}$
RegSrc$_{1:0}$
ALUControl$_{1:0}$
(a) Control Unit
Rd$_{3:0}$
PC Logic
PCS
Branch
CLK
Main FSM
RegW
MemW
IRWrite
NextPC
Register Enables
AdrSrc
ResultSrc$_{1:0}$
ALUSrcA
ALUSrcB$_{1:0}$
Multiplexer Selects
Op$_{1:0}$
5,0
Funct$_{5:0}$
4:0
ALUOp
ALU Decoder
ALUControl$_{1:0}$
FlagW$_{1:0}$
Op$_{1:0}$
Instr Decoder
ImmSrc$_{1:0}$
RegSrc$_{1:0}$
(b) Decoder
NextPC
PCS
RegW
MemW
FlagW$_{1:0}$
PCWrite
RegWrite
MemWrite
CLK
CondEx
Cond$_{3:0}$
CLK
[3:2]
Flags$_{3:2}$
[1]
FlagWrite$_{1:0}$
Condition Check
ALUFlags$_{3:0}$
CLK
[1:0]
Flags$_{1:0}$
[0]
(c) Conditional Logic

Figure 7.31 Multicycle control unit

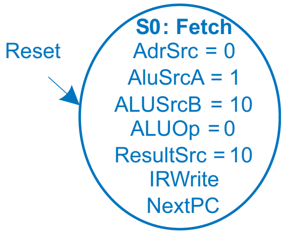

Reset
S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC

Figure 7.32 **Fetch**

Table 7.6 **Instr Decoder logic for *RegSrc* and *ImmSrc***

Instruction	*Op*	$Funct_5$	$Funct_0$	$RegSrc_1$	$RegSrc_0$	$ImmSrc_{1:0}$
`LDR`	01	X	1	X	0	01
`STR`	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
`B`	10	X	X	X	1	10

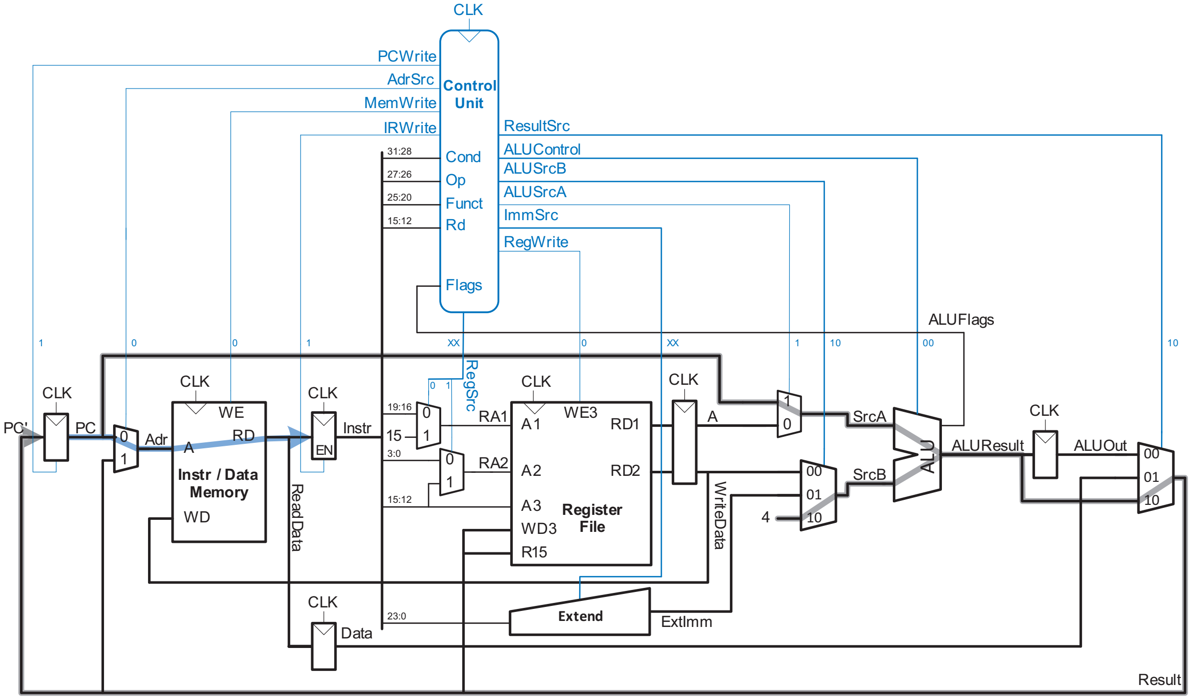

CLK
Control Unit
PCWrite
AdrSrc
MemWrite
IRWrite
ResultSrc
ALUControl
ALUSrcB
ALUSrcA
ImmSrc
RegWrite
Cond
Op
Funct
Rd
Flags
ALUFlags
31:28
27:26
25:20
15:12
RegSrc
PC'
PC
Adr
A
RD
WE
WD
Instr / Data Memory
ReadData
Instr
EN
Data
19:16
15
3:0
15:12
23:0
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
Extend
ExtImm
WriteData
SrcA
SrcB
4
ALU
ALUResult
ALUOut
Result
00
01
10

Figure 7.33 **Data flow during the fetch step**

use it to compute $PC + 4$ at the same time that it fetches the instruction. $ALUSrcA = 1$, so *SrcA* comes from the PC. $ALUSrcB = 10$, so *SrcB* is the constant 4. $ALUOp = 0$, so the ALU produces $ALUControl = 00$ to make the ALU add. To update the PC with $PC + 4$, $ResultSrc = 10$ to choose the *ALUResult* and $NextPC = 1$ to enable *PCWrite*.

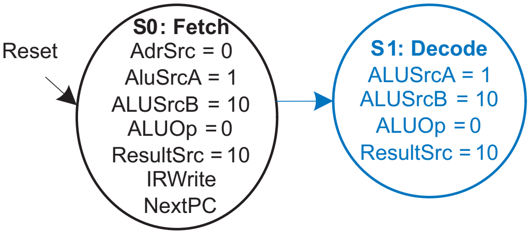


Figure 7.34 Decode

The second step is to read the register file and/or immediate and decode the instructions. The registers and immediate are selected based on *RegSrc* and *ImmSrc*, which are computed by the Instr Decoder based on *Instr*. $RegSrc_0$ should be 1 for branches to read $PC+8$ as *SrcA*. $RegSrc_1$ should be 1 for stores to read the store value as *SrcB*. *ImmSrc* should be 00 for data-processing instructions to select an 8-bit immediate, 01 for loads and stores to select a 12-bit immediate, and 10 for branches to select a 24-bit immediate. Because the multicycle FSM is a Moore machine whose outputs depend only on the current state, the FSM cannot directly produce these selects that depend on *Instr*. The FSM could be organized as a Mealy machine whose outputs depend on *Instr* as well as the state, but this would be messy. Instead, we choose the simplest solution, which is to make these selects combinational functions of *Instr*, as given in Table 7.6. Taking advantage of don't cares, the Instr Decoder logic can be simplified to:

$$RegSrc_1 = (Op == 01)$$

$$RegSrc_0 = (Op == 10)$$

$$ImmSrc_{1:0} = Op$$

Meanwhile, the ALU is reused to compute $PC+8$ by adding 4 more to the PC that was incremented in the Fetch step. Control signals are applied to select *PC* as the first ALU input ($ALUSrcA=1$) and 4 as the second input ($ALUSrcB=10$) and to perform addition ($ALUOp=0$). This sum is selected as the *Result* ($ResultSrc=10$) and provided to the *R15* input of the register file so that R15 reads as $PC+8$. The FSM Decode step is shown in Figure 7.34 and the data flow is shown in Figure 7.35, highlighting the R15 computation and the register file read.

Now the FSM proceeds to one of several possible states, depending on *Op* and *Funct* that are examined during the Decode step. If the instruction is a memory load or store (`LDR` or `STR`, $Op=01$), then the multicycle processor computes the address by adding the base address to the zero-extended offset. This requires $ALUSrcA=0$ to select the base address from the register file and $ALUSrcB=01$ to select *ExtImm*. $ALUOp=0$ so the ALU adds. The effective address is stored in the

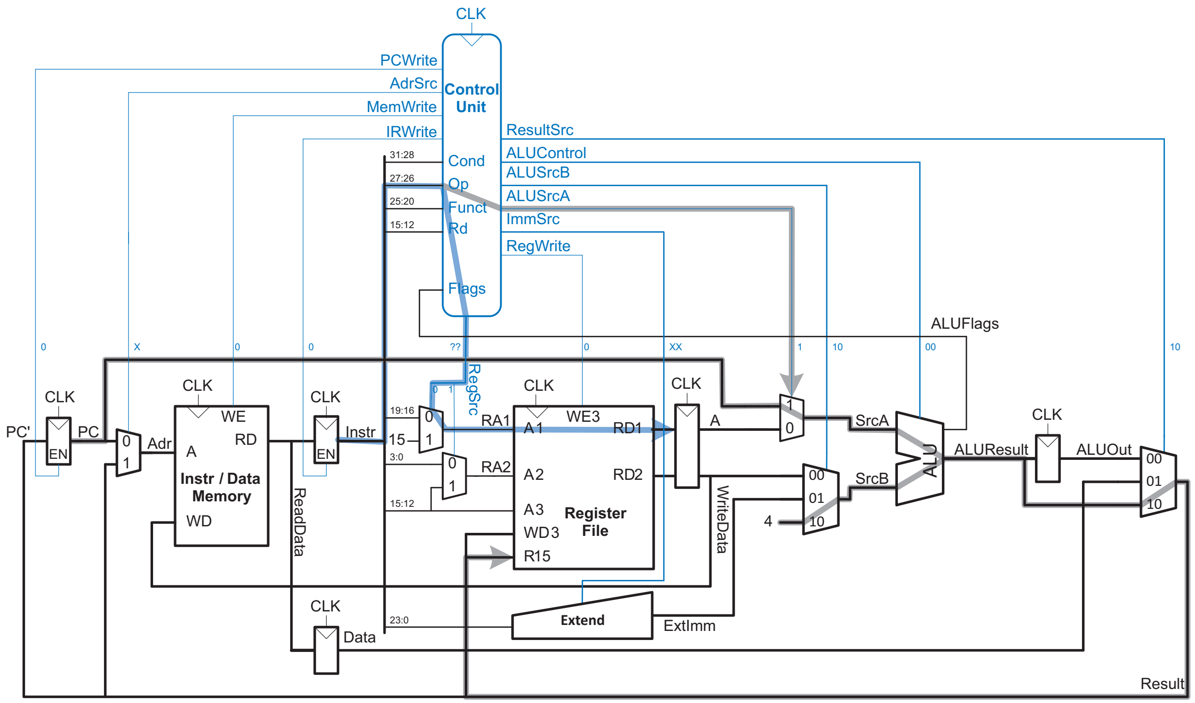

CLK
Control Unit
PCWrite
AdrSrc
MemWrite
IRWrite
31:28
27:26
25:20
15:12
Cond
Op
Funct
Rd
Flags
ResultSrc
ALUControl
ALUSrcB
ALUSrcA
ImmSrc
RegWrite
ALUFlags
0
X
0
0
??
0
XX
1
10
00
10
RegSrc
0 1
CLK
PC'
EN
PC
0
1
Adr
CLK
WE
A
RD
Instr / Data Memory
WD
CLK
EN
Instr
ReadData
19:16
15
3:0
0
1
0
1
15:12
RA1
RA2
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
CLK
A
WriteData
1
0
SrcA
00
01
10
4
SrcB
ALU
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
23:0
Extend
ExtImm
Result

Figure 7.35 **Data flow during the Decode step**

ALUOut register for use on the next step. The FSM MemAdr state is shown in Figure 7.36 and the data flow is highlighted in Figure 7.37.

If the instruction is `LDR` ($Funct_0 = 1$), then the multicycle processor must next read data from the memory and write it to the register file. These two steps are shown in Figure 7.38. To read from the memory, *ResultSrc* = 00 and *AdrSrc* = 1 to select the memory address that was just computed and saved in *ALUOut*. This address in memory is read and saved in the *Data* register during the MemRead step. Then, in the memory writeback step MemWB, *Data* is written to the register file. *ResultSrc* = 01 to choose *Result* from *Data* and *RegW* is asserted to write the register file, completing the `LDR` instruction. Finally, the FSM returns to the Fetch state to start the next instruction. For these and subsequent steps, try to visualize the data flow on your own.

From the MemAdr state, if the instruction is `STR` ($\text{Funct}_0 = 0$), the data read from the second port of the register file is simply written to memory. In this MemWrite state, *ResultSrc* = 00 and *AdrSrc* = 1 to select the address computed in the MemAdr state and saved in *ALUOut*. *MemW* is asserted to write the memory. Again, the FSM returns to the Fetch state. The state is shown in Figure 7.39.

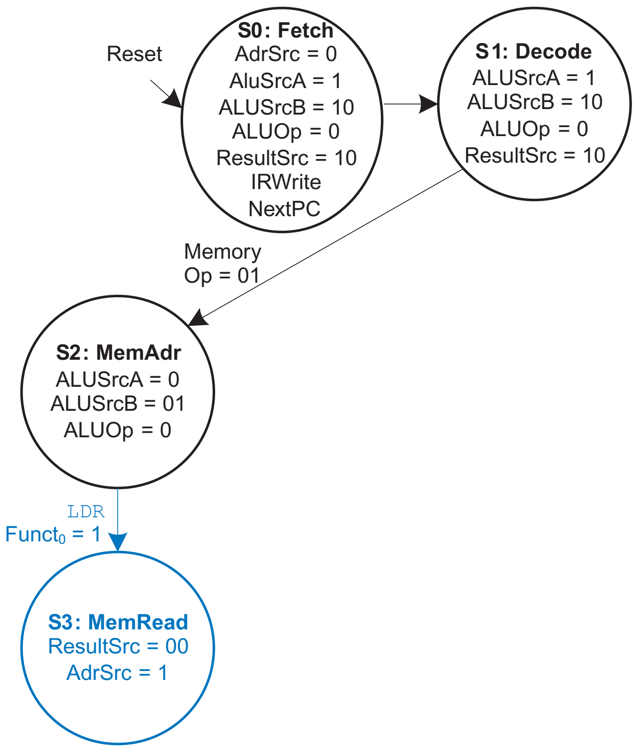

Reset
S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC
S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
Memory
Op = 01
S2: MemAdr
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
LDR
Funct$_0$ = 1
S3: MemRead
ResultSrc = 00
AdrSrc = 1

Figure 7.36 Memory address computation

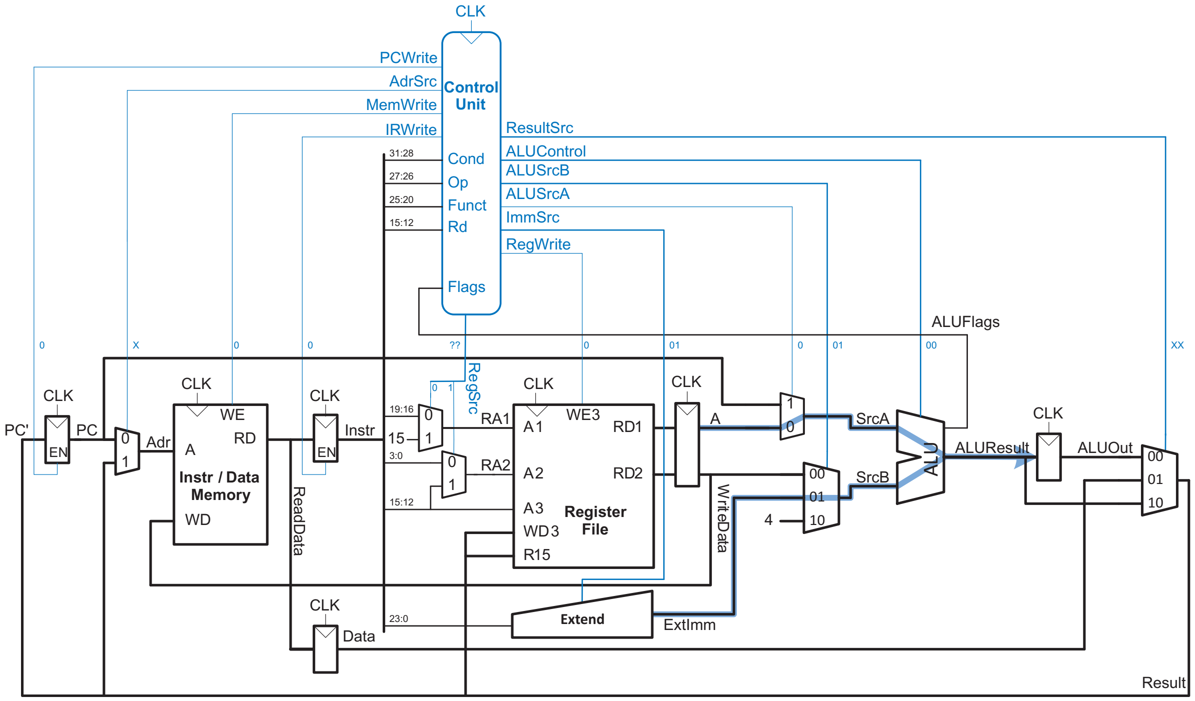

CLK
Control
Unit
PCWrite
AdrSrc
MemWrite
IRWrite
ResultSrc
ALUControl
ALUSrcB
ALUSrcA
ImmSrc
RegWrite
Cond
Op
Funct
Rd
Flags
31:28
27:26
25:20
15:12
ALUFlags
0
X
0
0
??
0
01
0
01
00
XX
RegSrc
0
1
CLK
PC'
PC
EN
0
1
Adr
CLK
WE
RD
A
Instr / Data
Memory
WD
CLK
EN
Instr
ReadData
19:16
15
3:0
15:12
0
1
0
1
RA1
RA2
CLK
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register
File
CLK
A
WriteData
1
0
00
01
10
4
SrcA
SrcB
ALU
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
23:0
Extend
ExtImm
Result

Figure 7.37 Data flow during memory address computation

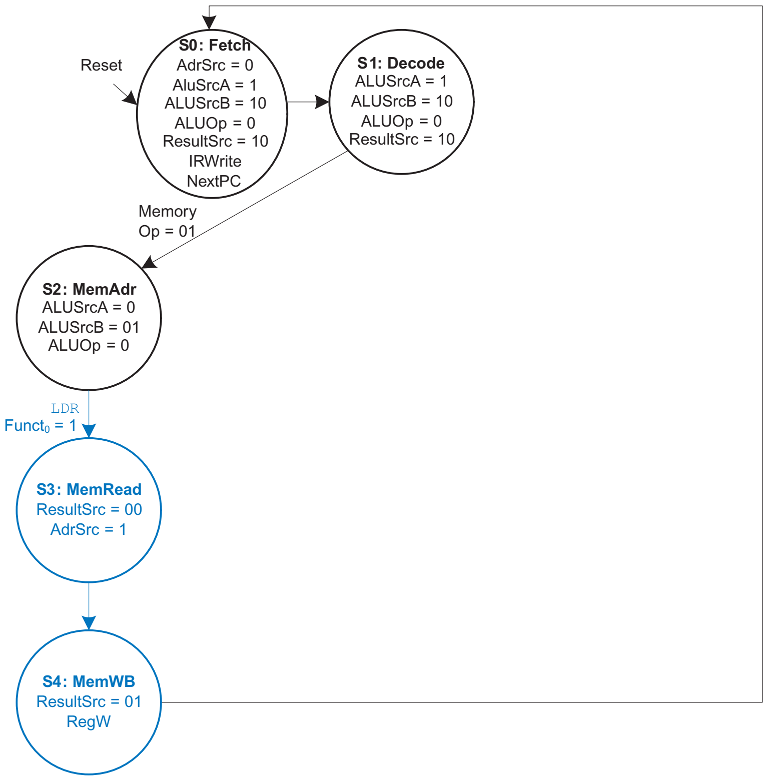

S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC
Reset
S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
Memory
Op = 01
S2: MemAdr
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
LDR
Funct0 = 1
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S4: MemWB
ResultSrc = 01
RegW

Figure 7.38 **Memory read**

For data-processing instructions ($Op = 00$), the multicycle processor must calculate the result using the ALU and write that result to the register file. The first source always comes from the register ($ALUSrcA = 0$). $ALUOp = 1$ so the ALU Decoder chooses the appropriate $ALUControl$ for the specific instruction based on *cmd* ($Funct_{4:1}$). The second source comes from the register file for register instructions ($ALUSrcB = 00$) or from *ExtImm* for immediate instructions ($ALUSrcB = 01$). Thus, the FSM needs ExecuteR and ExecuteI states to cover these two possibilities. In either case, the data-processing instruction advances to the ALU Writeback state (ALUWB), in which the result is selected from *ALUOut* ($ResultSrc = 00$) and written to the register file ($RegW = 1$). All of these states are shown in Figure 7.40.

For a branch instruction, the processor must calculate the destination address ($PC + 8 +$ offset) and write it to the PC. During the Decode state, $PC + 8$ was already computed and read from the register file onto *RD1*. Therefore, during the Branch state, the controller uses $ALUSrcA = 0$ to choose R15 ($PC + 8$), $ALUSrcB = 01$ to choose *ExtImm*, and $ALUOp = 0$

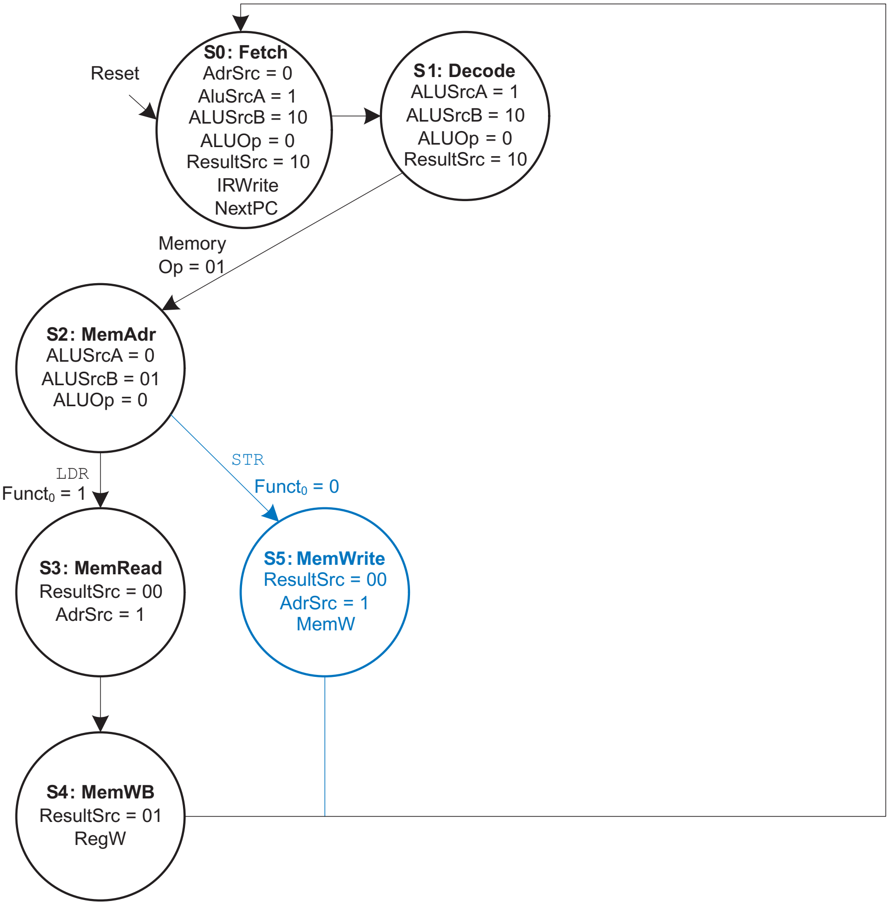

S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC
Reset
S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
Memory
Op = 01
S2: MemAdr
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
LDR
Funct₀ = 1
STR
Funct₀ = 0
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemW
S4: MemWB
ResultSrc = 01
RegW

Figure 7.39 **Memory write**

to add. The *Result* multiplexer chooses *ALUResult* (*ResultSrc* = 10). *Branch* is asserted to write the result to the PC.

Putting these steps together, Figure 7.41 shows the complete Main FSM state transition diagram for the multicycle processor. The function of each state is summarized below the figure. Converting the diagram to hardware is a straightforward but tedious task using the techniques of Chapter 3. Better yet, the FSM can be coded in an HDL and synthesized using the techniques of Chapter 4.

7.4.3 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor

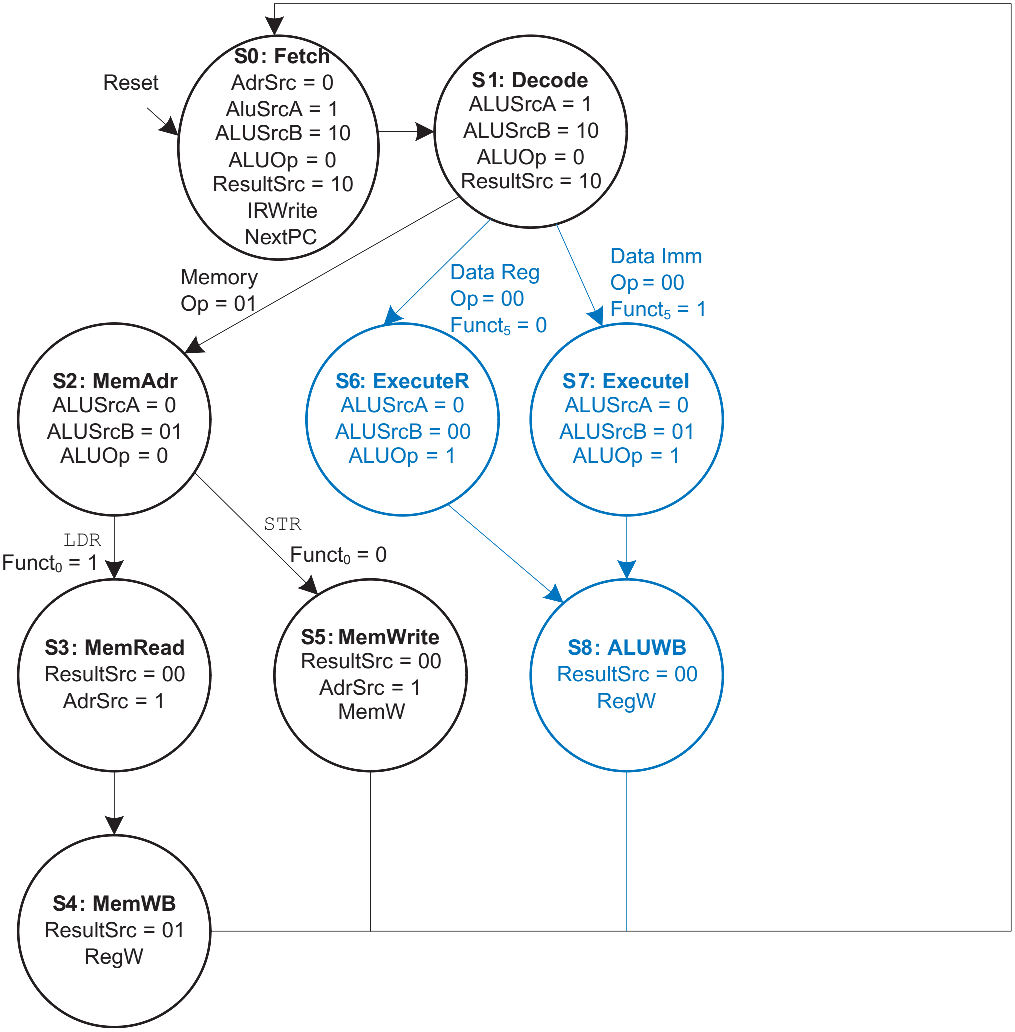
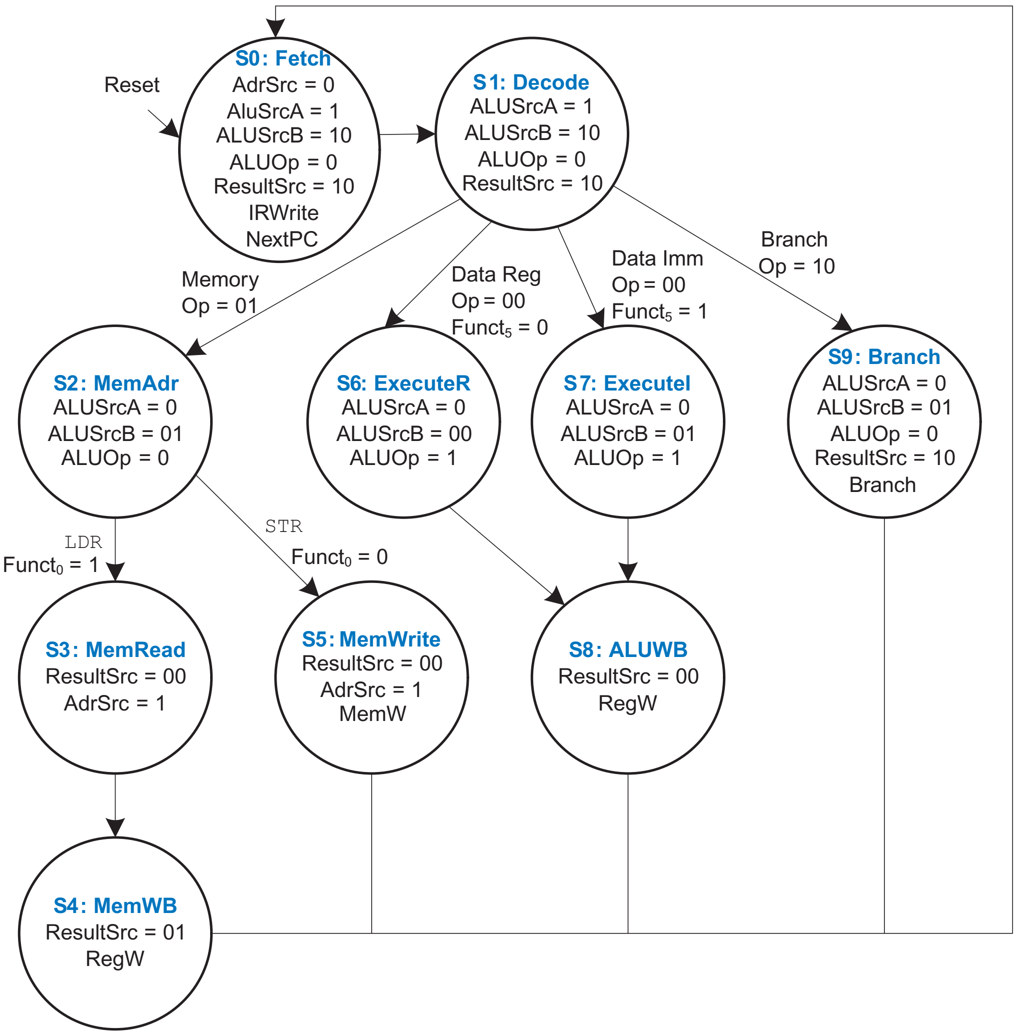

Reset
S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC
S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
Memory
Op = 01
Data Reg
Op = 00
Funct5 = 0
Data Imm
Op = 00
Funct5 = 1
S2: MemAdr
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
S6: ExecuteR
ALUSrcA = 0
ALUSrcB = 00
ALUOp = 1
S7: ExecuteI
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 1
LDR
Funct0 = 1
STR
Funct0 = 0
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemW
S8: ALUWB
ResultSrc = 00
RegW
S4: MemWB
ResultSrc = 01
RegW

Figure 7.40 Data-processing

performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for branches, four for data-processing instructions and stores, and five for loads. The CPI depends on the relative likelihood that each instruction is used.


Reset
S0: Fetch
AdrSrc = 0
AluSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
IRWrite
NextPC
S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 0
ResultSrc = 10
Memory
Op = 01
Data Reg
Op = 00
Funct$_5$ = 0
Data Imm
Op = 00
Funct$_5$ = 1
Branch
Op = 10
S2: MemAdr
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
S6: ExecuteR
ALUSrcA = 0
ALUSrcB = 00
ALUOp = 1
S7: ExecuteI
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 1
S9: Branch
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 0
ResultSrc = 10
Branch
LDR
Funct$_0$ = 1
STR
Funct$_0$ = 0
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemW
S8: ALUWB
ResultSrc = 00
RegW
S4: MemWB
ResultSrc = 01
RegW

State	Datapath µOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Figure 7.41 Complete multicycle control FSM

Example 7.5 MULTICYCLE PROCESSOR CPI

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 13% branches, and 52% data-processing instructions.[2] Determine the average CPI for this benchmark.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used. For this benchmark, average CPI = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12. This is better than the worst-case CPI of 5, which would be required if all instructions took the same time.

Recall that we designed the multicycle processor so that each cycle involved one ALU operation, memory access, or register file access. Let us assume that the register file is faster than the memory and that writing memory is faster than reading memory. Examining the datapath reveals two possible critical paths that would limit the cycle time:

1. From the PC through the *SrcA* multiplexer, ALU, and result multiplexer to the *R15* port of the register file to the *A* register
2. From *ALUOut* through the *Result* and *Adr* muxes to read memory into the *Data* register

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup} \tag{7.4}$$

The numerical values of these times will depend on the specific implementation technology.

Example 7.6 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle is wondering whether the multicycle processor would be faster than the single-cycle processor. For both designs, he plans on using the 16-nm CMOS manufacturing process with the delays given in Table 7.5. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.5).

Solution: According to Equation 7.4, the cycle time of the multicycle processor is $T_{c2} = 40 + 2(25) + 200 + 50 = 340$ ps. Using the CPI of 4.12 from Example 7.5, the total execution time is $T_2 = (100 \times 10^9$ instructions)(4.12 cycles/instruction)$(340 \times 10^{-12}$ s/cycle) = 140 seconds. According to Example 7.4, the single-cycle processor had a total execution time of 84 seconds.

---

[2] Data from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011.

One of the original motivations for building a multicycle processor was to avoid making all instructions take as long as the slowest one. Unfortunately, this example shows that the multicycle processor is slower than the single-cycle processor given the assumptions of CPI and circuit element delays. The fundamental problem is that even though the slowest instruction, `LDR`, was broken into five steps, the multicycle processor cycle time was not nearly improved five-fold. This is partly because not all of the steps are exactly the same length, and partly because the 90-ps sequencing overhead of the register clock-to-Q and setup time must now be paid on every step, not just once for the entire instruction. In general, engineers have learned that it is difficult to exploit the fact that some computations are faster than others unless the differences are large.

Compared with the single-cycle processor, the multicycle processor is likely to be less expensive because it shares a single memory for instructions and data and because it eliminates two adders. It does, however, require five nonarchitectural registers and additional multiplexers.

7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five-times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*. They are similar to the five steps that the multicycle processor used to perform `LDR`. In the *Fetch* stage, the processor reads the instruction from instruction memory. In the *Decode* stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the *Execute* stage, the processor performs a computation with the ALU. In the *Memory* stage, the processor reads or writes data memory. Finally, in the *Writeback* stage, the processor writes the result to the register file, when applicable.

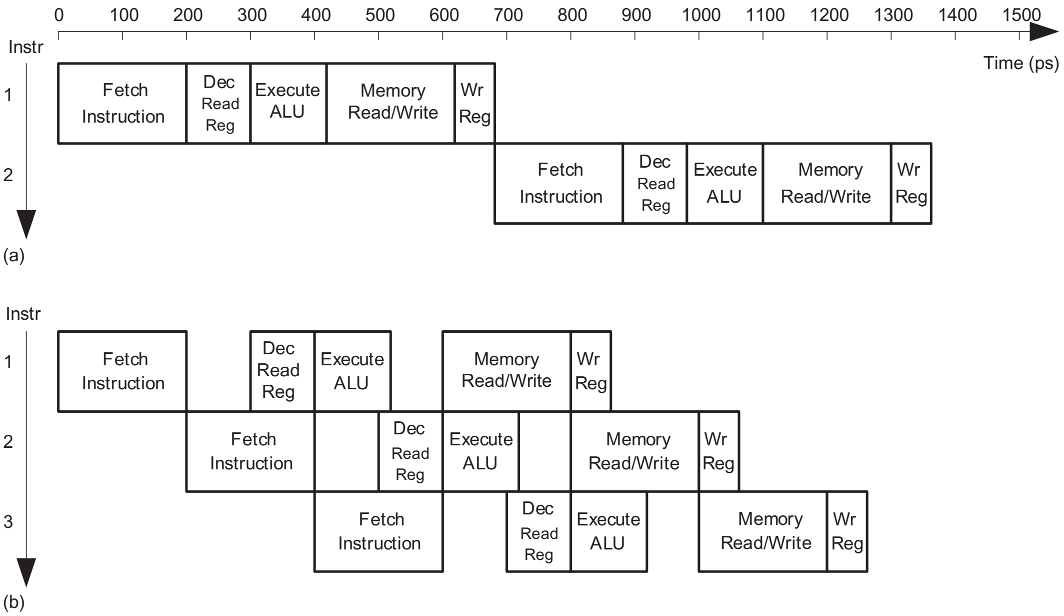


Figure 7.42 **Timing diagrams: (a) single-cycle processor and (b) pipelined processor**

Figure 7.42 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis, and instructions are on the vertical axis. The diagram assumes the logic element delays from Table 7.5 but ignores the delays of multiplexers and registers. In the single-cycle processor (Figure 7.42(a)), the first instruction is read from memory at time 0; next, the operands are read from the register file; and, then, the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file by 680 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of $200 + 100 + 120 + 200 + 60 = 680$ ps and a throughput of 1 instruction per 680 ps (1.47 billion instructions per second).

In the pipelined processor (Figure 7.42(b)), the length of a pipeline stage is set at 200 ps by the slowest stage, the memory access (in the Fetch or Memory stage). At time 0, the first instruction is fetched from memory. At 200 ps, the first instruction enters the Decode stage, and a second instruction is fetched. At 400 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched. And so forth, until all the instructions complete. The instruction latency is $5 \times 200 = 1000$ ps. The throughput is 1 instruction per 200 ps (5 billion instructions per second). Because the stages are not perfectly balanced with equal amounts of logic, the latency is longer for the pipelined

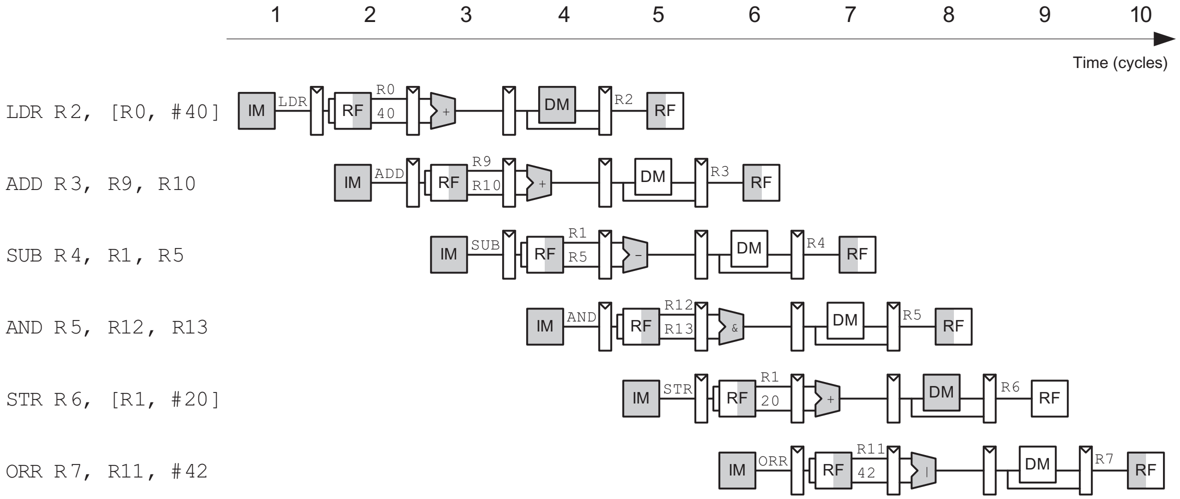


Figure 7.43 **Abstract view of pipeline in operation**

processor than for the single-cycle processor. Similarly, the throughput is not quite five-times as great for a five-stage pipeline as for the single-cycle processor. Nevertheless, the throughput advantage is substantial.

Figure 7.43 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file writeback—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycles in which a particular instruction is in each stage. For example, the `SUB` instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the `ORR` instruction is being fetched from instruction memory, whereas R1 is being read from the register file, the ALU is computing R12 AND R13, the data memory is idle, and the register file is writing a sum to R3. Stages are shaded to indicate when they are used. For example, the data memory is used by `LDR` in cycle 4 and by `STR` in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by every instruction except `STR`. In the pipelined processor, the register file is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written and read back within a single cycle.

A central challenge in pipelined systems is handling hazards that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed. For example, if the `ADD` in Figure 7.43 used R2 rather than R10, a hazard would occur because the R2 register has not been written by the `LDR` by the time it is read by the `ADD`. After designing the pipelined datapath and control, this section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards. Finally, this section revisits performance analysis considering sequencing overhead and the impact of hazards.

7.5.1 Pipelined Datapath

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers.

Figure 7.44(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.44(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage, but the write address and data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in Section 7.5.3. The register file in the pipelined processor writes on the falling edge of CLK so that it

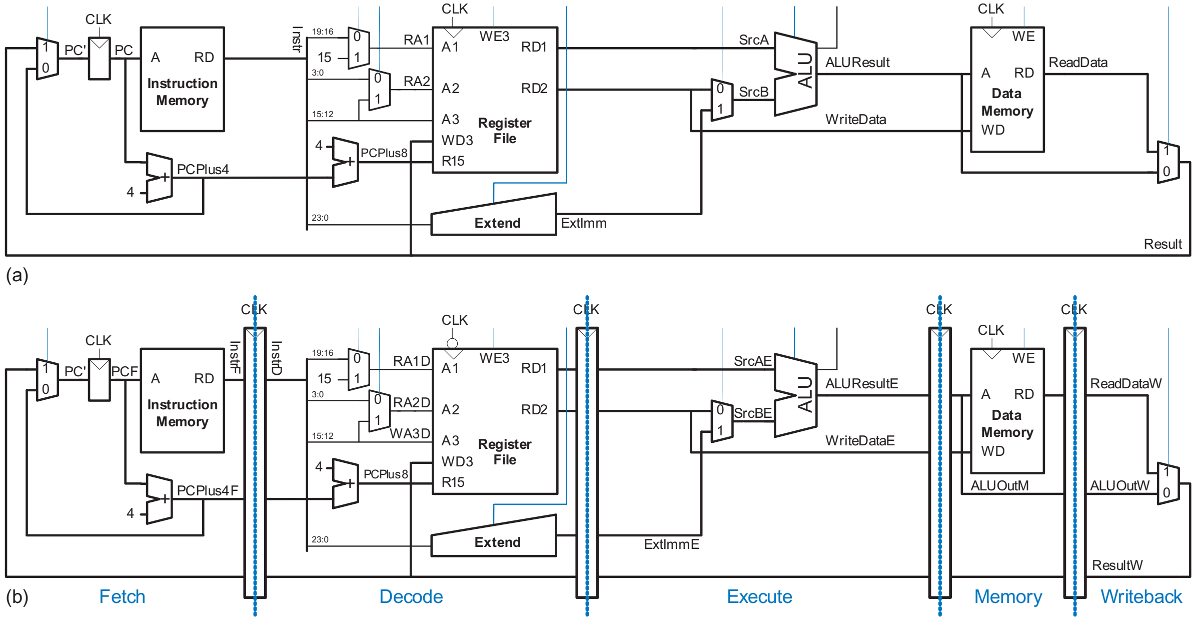

CLK
PC'
PC
A RD
Instruction Memory
Instr
19:16
15
3:0
15:12
23:0
RA1
RA2
A1
A2
A3
WE3
RD1
RD2
Register File
WD3
R15
4
PCPlus8
PCPlus4
Extend
ExtImm
SrcA
SrcB
ALU
ALUResult
WriteData
WE
A RD
Data Memory
WD
ReadData
Result
(a)
InstrF
InstrD
PCF
PCPlus4F
RA1D
RA2D
WA3D
SrcAE
SrcBE
ALUResultE
WriteDataE
ExtImmE
ALUOutM
ReadDataW
ALUOutW
ResultW
(b)
Fetch
Decode
Execute
Memory
Writeback

Figure 7.44 Datapaths: (a) single-cycle and (b) pipelined

can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.

One of the subtle but critical issues in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison. Figure 7.44(b) has an error related to this issue. Can you find it?

The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from *ResultW*, a Writeback stage signal. But the write address comes from $InstrD_{15:12}$ (also known as *WA3D*), which is a Decode stage signal. In the pipeline diagram of Figure 7.43, during cycle 5, the result of the `LDR` instruction would be incorrectly written to R5 rather than R2.

Figure 7.45 shows a corrected datapath, with the modification in black. The *WA3* signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. *WA3W* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may note that the *PC'* logic is also problematic, because it might be updated with a Fetch or a Writeback stage signal (*PCPlus4F* or *ResultW*). This control hazard will be fixed in Section 7.5.3.

Figure 7.46 shows another optimization to save a 32-bit adder and register in the PC logic. Observe in Figure 7.45 that each time the program counter is incremented, *PCPlus4F* is simultaneously written to the PC and the pipeline register between the Fetch and Decode stages. Moreover, on the subsequent cycle, the value in both of these registers is incremented by 4 again. Thus, *PCPlus4F* for the instruction in the Fetch stage is logically equivalent to *PCPlus8D* for the instruction in the Decode stage. Sending this signal ahead saves the pipeline register and second adder.[3]

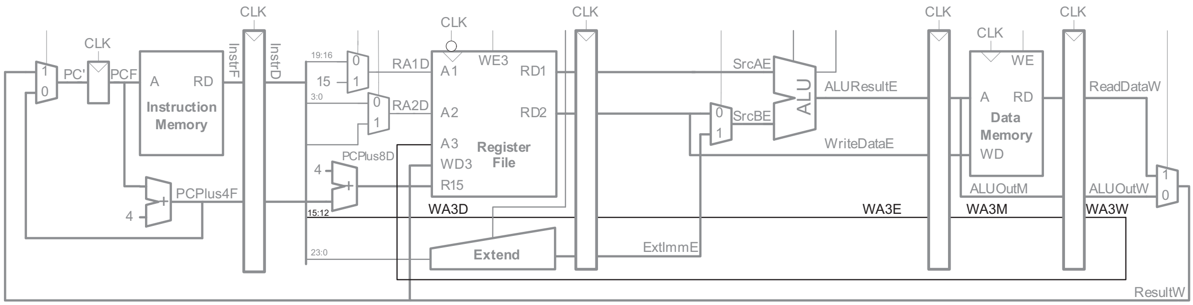

CLK
PC'
PCF
A
RD
Instruction Memory
4
PCPlus4F
CLK
InstrF
InstrD
19:16
15
3:0
0
1
RA1D
RA2D
PCPlus8D
15:12
23:0
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
WA3D
Extend
CLK
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
ExtImmE
WA3E
CLK
CLK
WE
A
RD
Data Memory
WD
ALUOutM
WA3M
CLK
ReadDataW
ALUOutW
WA3W
ResultW

Figure 7.45 Corrected pipelined datapath

---

[3] There is a potential problem with this simplification when the PC is written with *ResultW* rather than *PCPlus4F*. However, this case is handled in Section 7.5.3 by flushing the pipeline, so *PCPlus8D* becomes a don't care and the pipeline still operates correctly.

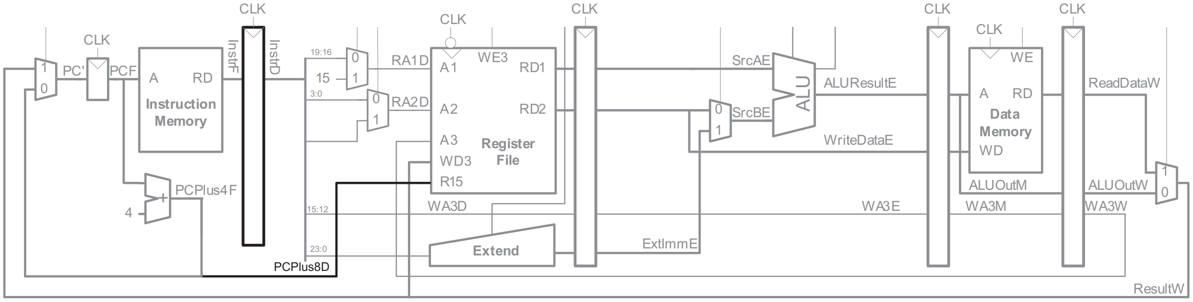

CLK
PC'
PCF
A
RD
Instruction Memory
PCPlus4F
4
InstrF
InstrD
19:16
15
3:0
0
1
RA1D
RA2D
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register File
WA3D
15:12
23:0
PCPlus8D
Extend
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
WA3E
ExtImmE
WE
Data Memory
WD
ALUOutM
WA3M
ReadDataW
ALUOutW
WA3W
ResultW

Figure 7.46 Optimized PC logic eliminating a register and adder

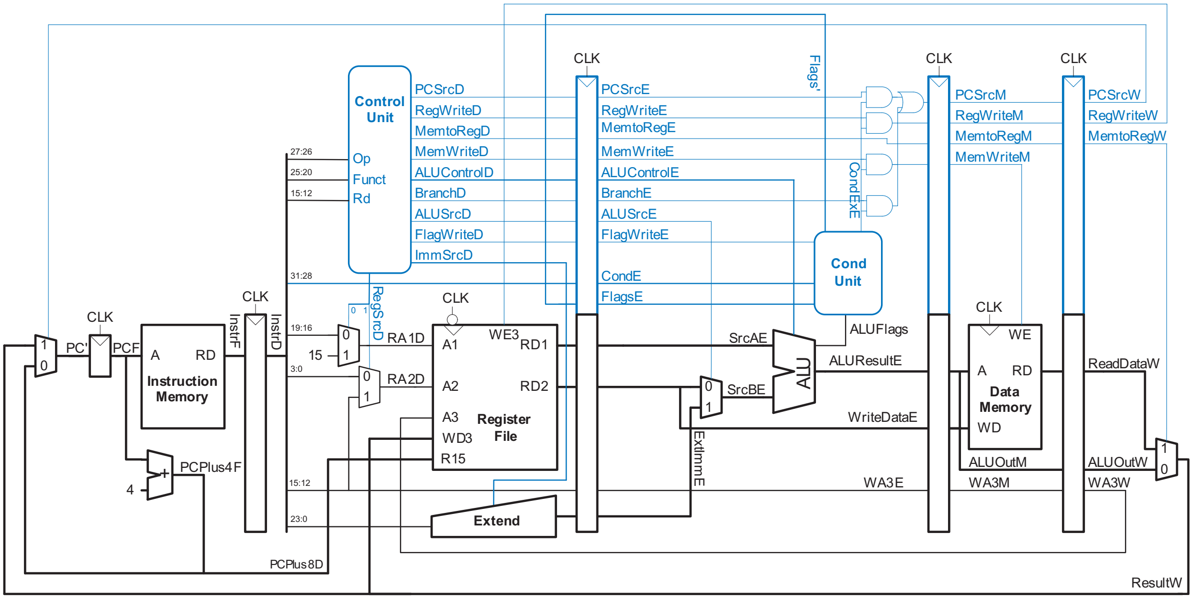

Control Unit
Op
Funct
Rd
27:26
25:20
15:12
31:28
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
BranchD
ALUSrcD
FlagWriteD
ImmSrcD
RegSrcD
CLK
PCSrcE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
BranchE
ALUSrcE
FlagWriteE
CondE
FlagsE
Flags'
CondExE
Cond Unit
ALUFlags
PCSrcM
RegWriteM
MemtoRegM
MemWriteM
PCSrcW
RegWriteW
MemtoRegW
PC'
PCF
A
RD
Instruction Memory
PCPlus4F
4
InstrF
InstrD
19:16
15
3:0
0
1
RA1D
RA2D
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register File
15:12
23:0
PCPlus8D
Extend
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
ExtImmE
WA3E
WE
Data Memory
WD
ALUOutM
WA3M
ReadDataW
ALUOutW
WA3W
ResultW

Figure 7.47 Pipelined processor with control

7.5.2 Pipelined Control

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit. The control unit examines the *Op* and *Funct* fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.2. These control signals must be pipelined along with the data so that they remain synchronized with the instruction. The control unit also examines the *Rd* field to handle writes to R15 (PC).

The entire pipelined processor with control is shown in Figure 7.47. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *WA3* was pipelined in Figure 7.45.

7.5.3 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has not yet completed, a *hazard* occurs.

The register file can be read and written in the same cycle. The write takes place during the first half of the cycle and the read takes place during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.48 illustrates hazards that occur when one instruction writes a register (R1) and subsequent instructions read this register. This is called a *read after write* (*RAW*) *hazard*. The `ADD` instruction writes a result into R1 in the first half of cycle 5. However, the `AND` instruction reads R1 on cycle 3, obtaining the wrong value. The `ORR` instruction reads R1 on cycle 4, again obtaining the wrong value. The `SUB` instruction reads R1 in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of R1. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions reads that register. Without special treatment, the pipeline will compute the wrong result.

A software solution would be to require the programmer or compiler to insert `NOP` instructions between the `ADD` and `AND` instructions so that the dependent instruction does not read the result (R1) until it is available in the register file, as shown in Figure 7.49. Such a *software interlock* complicates programming as well as degrading performance, so it is not ideal.

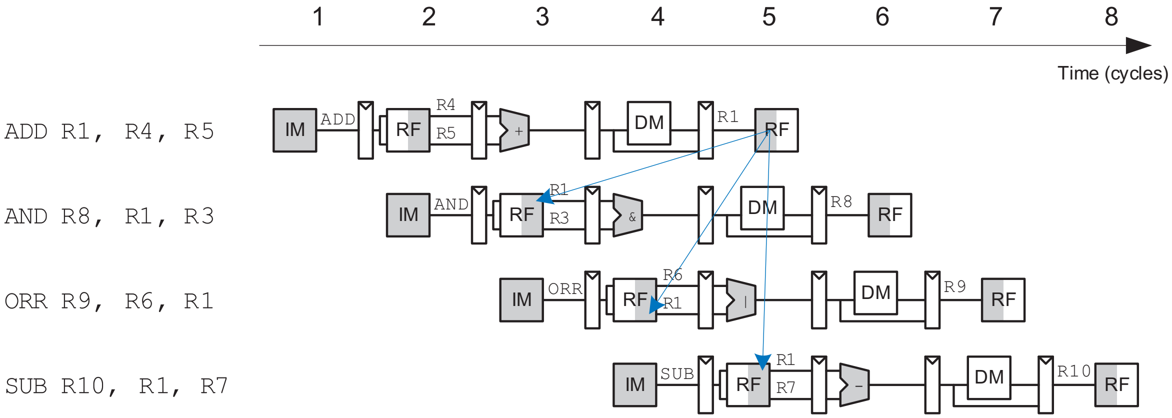


Figure 7.48 Abstract pipeline diagram illustrating hazards

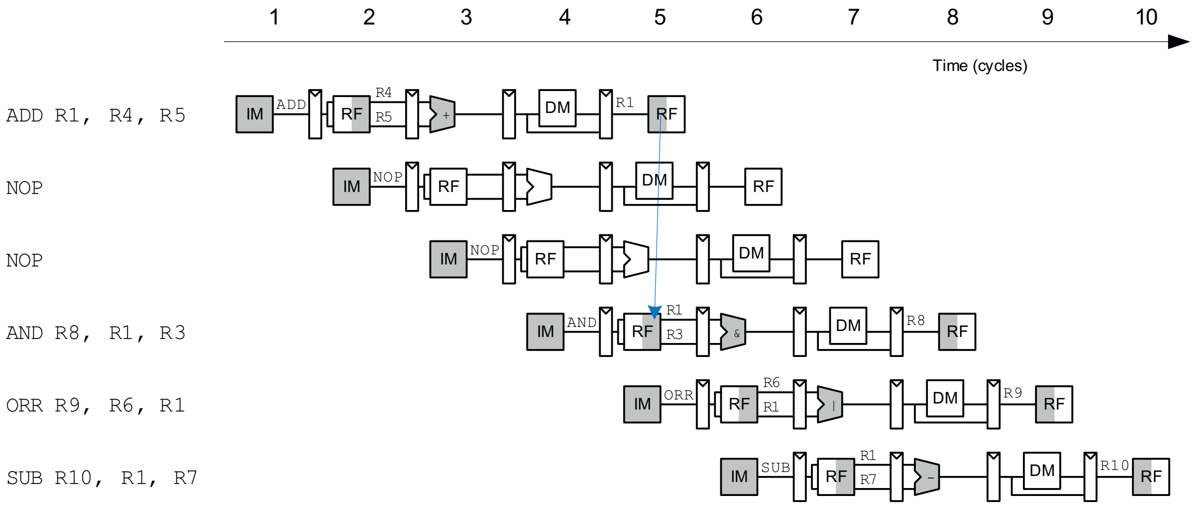

1
2
3
4
5
6
7
8
9
10
Time (cycles)
ADD R1, R4, R5
NOP
NOP
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
ADD
RF
R4
R5
+
DM
R1
RF
NOP
RF
DM
RF
NOP
RF
DM
RF
AND
RF
R1
R3
&
DM
R8
RF
ORR
RF
R6
R1
|
DM
R9
RF
SUB
RF
R1
R7
-
DM
R10
RF

Figure 7.49 **Solving data hazard with** `NOP`

On closer inspection, observe from Figure 7.48 that the sum from the `ADD` instruction is computed by the ALU in cycle 3 and is not strictly needed by the `AND` instruction until the ALU uses it in cycle 4. In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without waiting for the result to appear in the register file and without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be produced before the subsequent instruction uses the result. In any event, something must be done to solve hazards so that the program executes correctly despite the pipelining.

Hazards are classified as data hazards or control hazards. A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In the remainder of this section, we enhance the pipelined processor with a Hazard Unit that detects hazards and handles them appropriately, so that the processor executes the program correctly.

Solving Data Hazards with Forwarding

Some data hazards can be solved by *forwarding* (also called *bypassing*) a result from the Memory or Writeback stage to a dependent instruction in

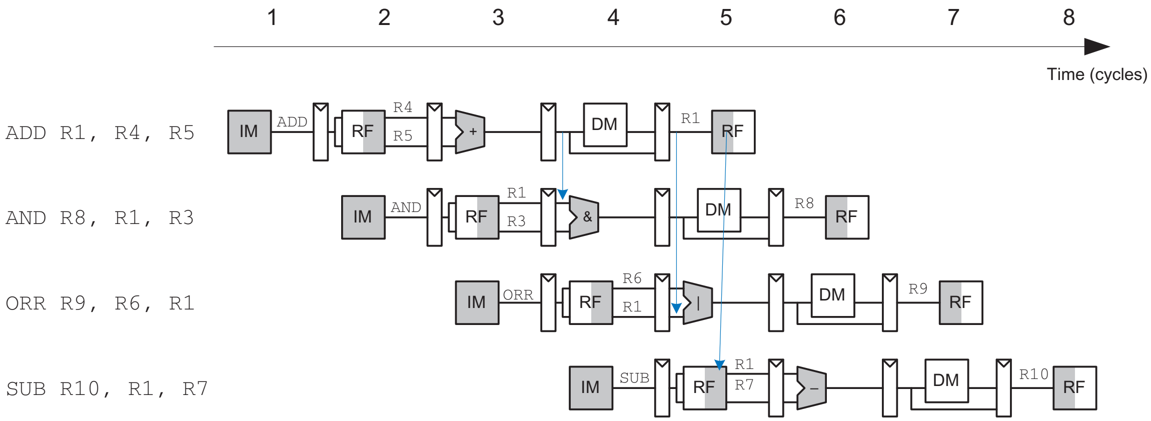


Figure 7.50 **Abstract pipeline diagram illustrating forwarding**

the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Figure 7.50 illustrates this principle. In cycle 4, R1 is forwarded from the Memory stage of the `ADD` instruction to the Execute stage of the dependent `AND` instruction. In cycle 5, R1 is forwarded from the Writeback stage of the `ADD` instruction to the Execute stage of the dependent `ORR` instruction.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.51 modifies the pipelined processor to support forwarding. It adds a *Hazard Unit* and two *forwarding multiplexers*. The Hazard Unit receives four match signals from the datapath (abbreviated to *Match* in Figure 7.51) that indicate whether the source registers in the Execute stage match the destination registers in the Memory and Execute stages:

```
Match_1E_M = (RA1E == WA3M)
Match_1E_W = (RA1E == WA3W)
Match_2E_M = (RA2E == WA3M)
Match_2E_W = (RA2E == WA3W)
```

The Hazard Unit also receives the *RegWrite* signals from the Memory and Writeback stages to know whether the destination register will actually be written (e.g., the `STR` and `B` instructions do not write results to the register file and, hence, do not need to have their results forwarded).

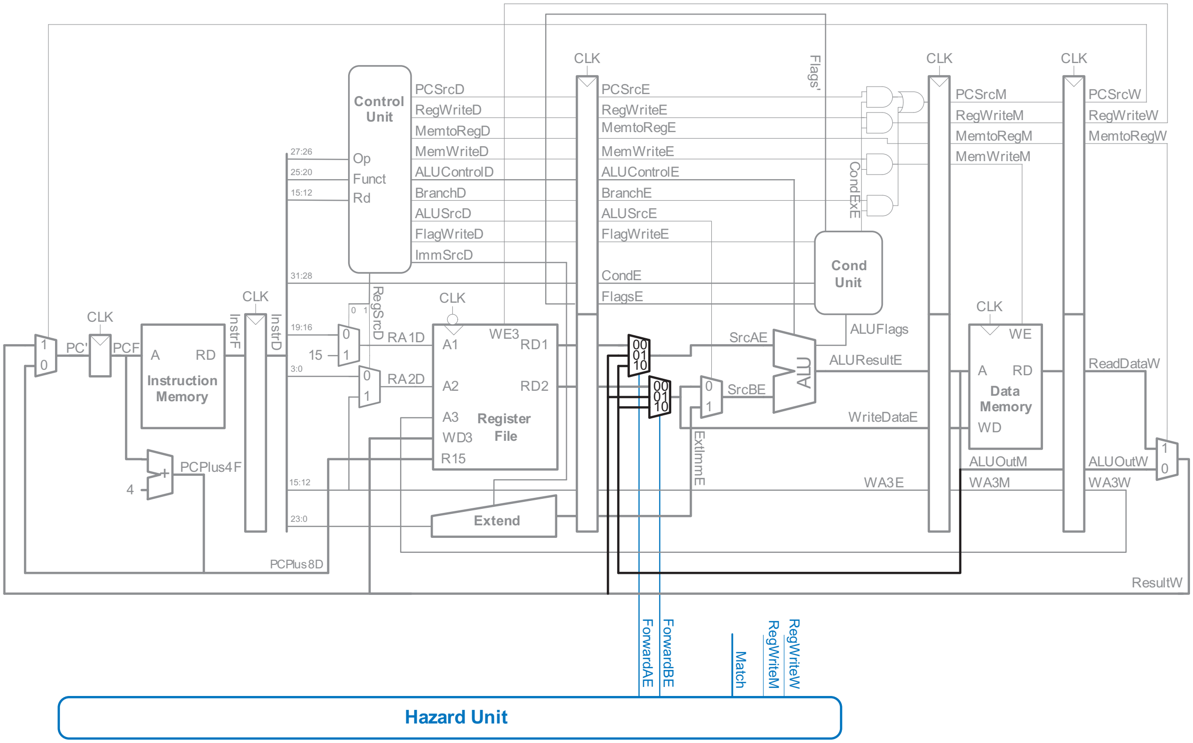

CLK
Control Unit
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
BranchD
ALUSrcD
FlagWriteD
ImmSrcD
Op
Funct
Rd
27:26
25:20
15:12
31:28
19:16
3:0
15:12
23:0
RegSrcD
RA1D
RA2D
15
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
Extend
PCPlus8D
InstrF
InstrD
PC'
PCF
A RD
Instruction Memory
PCPlus4F
4
PCSrcE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
BranchE
ALUSrcE
FlagWriteE
CondE
FlagsE
Flags'
CondExE
Cond Unit
ALUFlags
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
WA3E
ExtImmE
00
01
10
PCSrcM
RegWriteM
MemtoRegM
MemWriteM
WE
A RD
Data Memory
WD
ALUOutM
WA3M
PCSrcW
RegWriteW
MemtoRegW
ReadDataW
ALUOutW
WA3W
ResultW
ForwardAE
ForwardBE
Match
RegWriteM
RegWriteW
Hazard Unit

Figure 7.51 **Pipelined processor with forwarding to solve hazards**

Note that these signals are *connected by name*. In other words, rather than cluttering up the diagram with long wires running from the control signals at the top to the Hazard Unit at the bottom, the connections are indicated by a short stub of wire labeled with the control signal name to which it is connected. The Match signal logic and pipeline registers for *RA1E* and *RA2E* are also left out to limit clutter.

The Hazard Unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage (*ALUOutM* or *ResultW*). It should forward from a stage if that stage will write a destination register and the destination register matches the source register. If both the Memory and Writeback stages contain matching destination registers, then the Memory stage should have priority, because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcAE* is given here. The forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Match_2E*.

```
if      (Match_1E_M • RegWriteM) ForwardAE = 10;  // SrcAE = ALUOutM
else if (Match_1E_W • RegWriteW) ForwardAE = 01;  // SrcAE = ResultW
else                             ForwardAE = 00;  // SrcAE from regfile
```

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the `LDR` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `LDR` instruction has a *two-cycle latency*, because a dependent instruction cannot use its result until two cycles later. Figure 7.52 shows this problem. The `LDR` instruction receives data from memory at the end of cycle 4. But the `AND` instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

The alternative solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.53 shows stalling the dependent instruction (`AND`) in the Decode stage. `AND` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`ORR`) must remain in the Fetch stage during both cycles as well, because the Decode stage is full.

In cycle 5, the result can be forwarded from the Writeback stage of `LDR` to the Execute stage of `AND`. Also in cycle 5, source R1 of the `ORR` instruction is read directly from the register file, with no need for forwarding.

Note that the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a *bubble*, and it behaves like

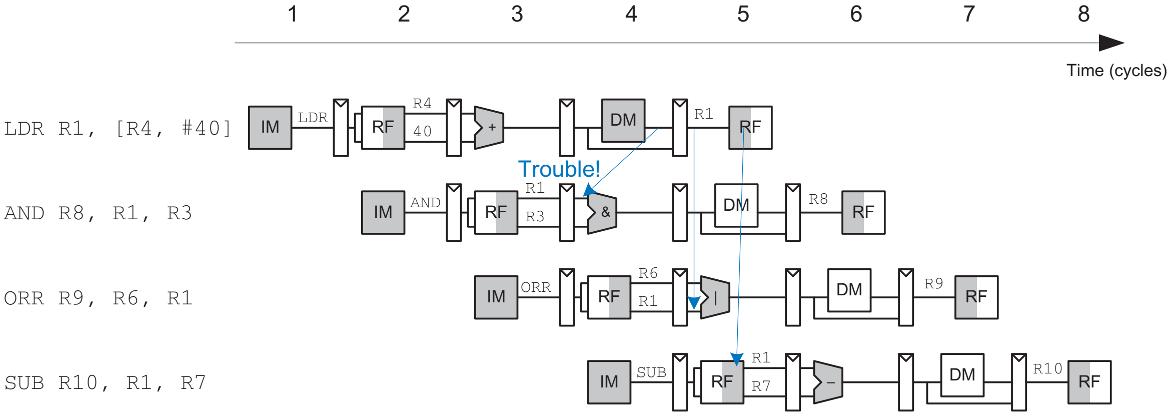


Figure 7.52 **Abstract pipeline diagram illustrating trouble forwarding from** `LDR`

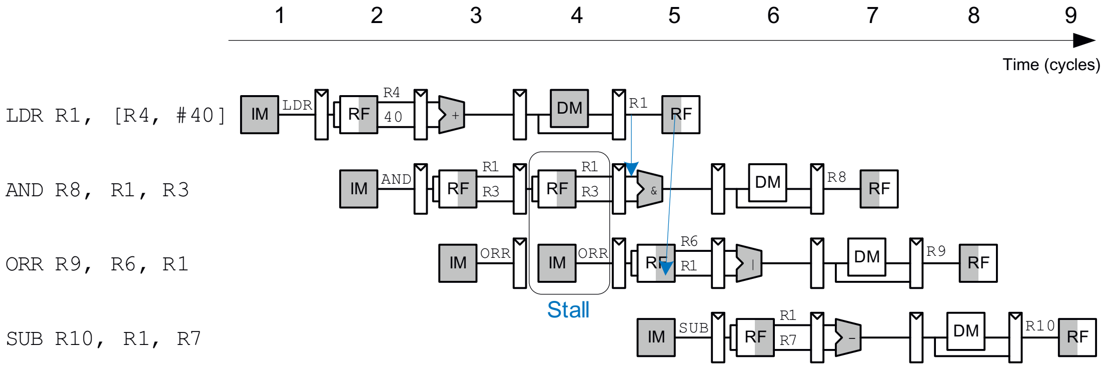


Figure 7.53 **Abstract pipeline diagram illustrating stall to solve hazards**

a `NOP` instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

Figure 7.54 modifies the pipelined processor to add stalls for `LDR` data dependencies. The Hazard Unit examines the instruction in the Execute stage. If it is an `LDR` and its destination register (*WA3E*) matches either source operand of the instruction in the Decode stage (*RA1D* or *RA2D*), then that instruction must be stalled in the Decode stage until the source operand is ready.

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When an `LDR` stall occurs, *StallD* and *StallF* are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble.

The *MemtoReg* signal is asserted for the `LDR` instruction. Hence, the logic to compute the stalls and flushes is

```
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E)
LDRstall = Match_12D_E • MemtoRegE
StallF = StallD = FlushE = LDRstall
```

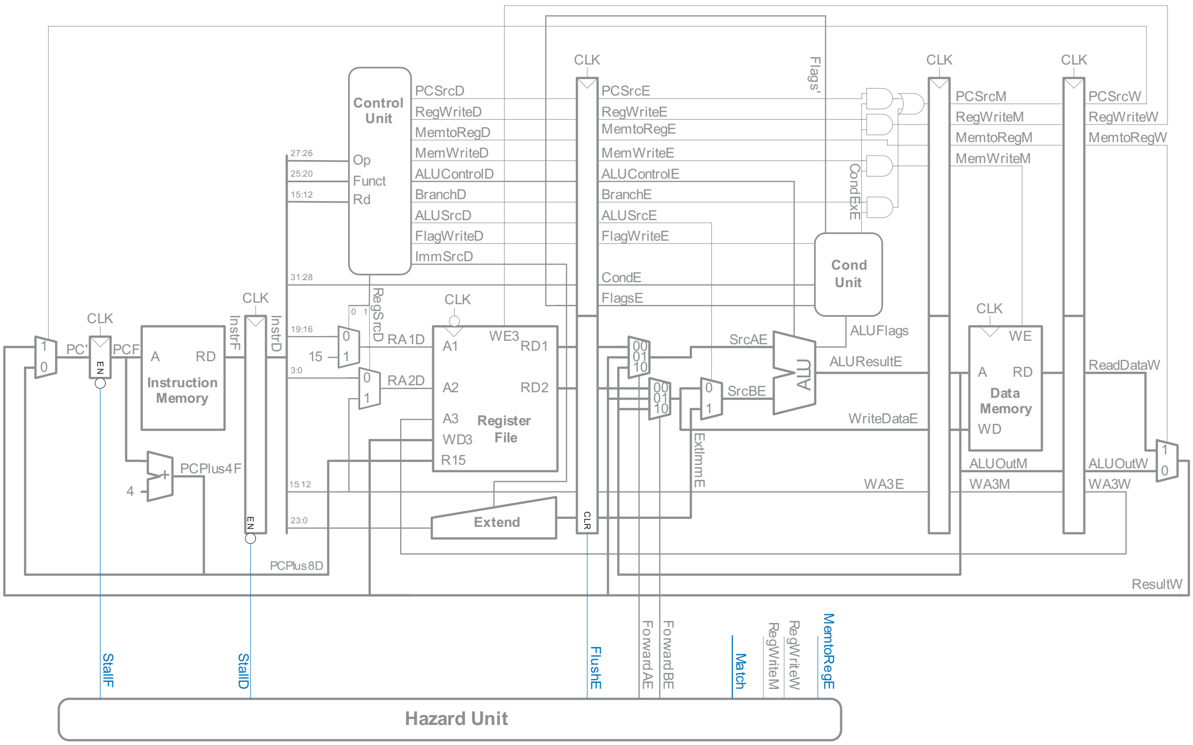

CLK
PC'
PCF
1
0
EN
A
RD
Instruction Memory
4
PCPlus4F
InstrF
InstrD
CLK
Control Unit
Op
Funct
Rd
27:26
25:20
15:12
31:28
19:16
3:0
15:12
23:0
15
RegSrcD
RA1D
RA2D
CLK
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register File
Extend
PCPlus8D
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
BranchD
ALUSrcD
FlagWriteD
ImmSrcD
CLK
CLR
PCSrcE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
BranchE
ALUSrcE
FlagWriteE
CondE
FlagsE
Flags'
CondExE
Cond Unit
ALUFlags
00
01
10
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
ExtImmE
WA3E
CLK
PCSrcM
RegWriteM
MemtoRegM
MemWriteM
CLK
WE
A
RD
Data Memory
WD
ALUOutM
WA3M
CLK
PCSrcW
RegWriteW
MemtoRegW
ReadDataW
ALUOutW
WA3W
ResultW
StallF
StallD
FlushE
ForwardAE
ForwardBE
Match
RegWriteM
RegWriteW
MemtoRegE
Hazard Unit

Figure 7.54 Pipelined processor with stalls to solve `LDR` data hazard

Solving Control Hazards

The `B` instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. Writes to R15 (PC) present a similar control hazard.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcW* is computed). Because the decision is made in the Writeback stage, the pipeline would have to be stalled for four cycles at every branch. This would severely degrade the system performance if it occurs often.

An alternative is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In the pipeline presented so far (Figure 7.54), the processor predicts that branches are not taken and simply continues executing the program in order until *PCSrcW* is asserted to select the next *PC* from *ResultW* instead. If the branch should have been taken, then the

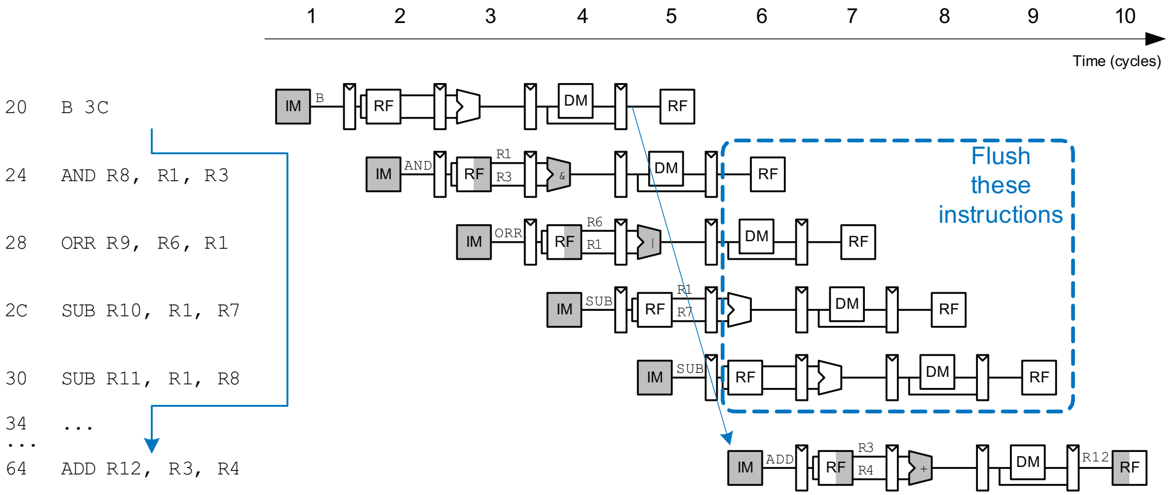


Figure 7.55 Abstract pipeline diagram illustrating flushing when a branch is taken

four instructions following the branch must be *flushed* (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the *branch misprediction penalty*.

Figure 7.55 shows such a scheme in which a branch from address 0x20 to address 0x64 is taken. The PC is not written until cycle 5, by which point the `AND`, `ORR`, and both `SUB` instructions at addresses 0x24, 0x28, 0x2C, and 0x30 have already been fetched. These instructions must be flushed, and the `ADD` instruction is fetched from address 0x64 in cycle 6. This is somewhat of an improvement, but flushing so many instructions when the branch is taken still degrades performance.

We could reduce the branch misprediction penalty if the branch decision could be made earlier. Observe that the branch decision can be made in the Execute stage when the destination address has been computed and *CondEx* is known. Figure 7.56 shows the pipeline operation with the early branch decision being made in cycle 3. In cycle 4, the `AND` and `ORR` instructions are flushed and the `ADD` instruction is fetched. Now the branch misprediction penalty is reduced to only two instructions rather than four.

Figure 7.57 modifies the pipelined processor to move the branch decision earlier and handle control hazards. A branch multiplexer is added before the PC register to select the branch destination from *ALUResultE*. The *BranchTakenE* signal controlling this multiplexer is asserted on branches whose condition is satisfied. *PCSrcW* is now only asserted for writes to the PC, which still occur in the Writeback stage.

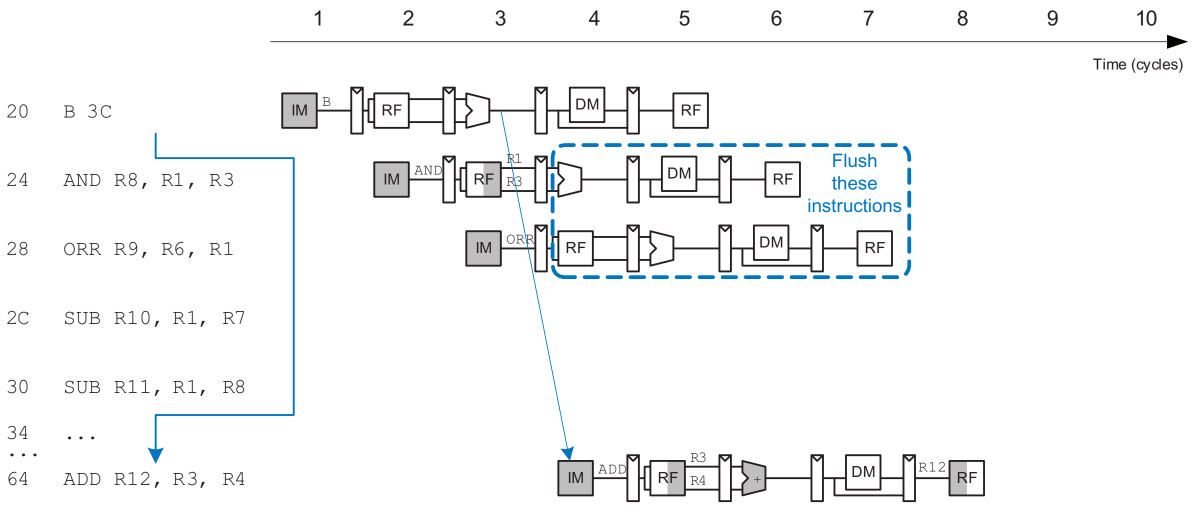

1
2
3
4
5
6
7
8
9
10
Time (cycles)
20 B 3C
24 AND R8, R1, R3
28 ORR R9, R6, R1
2C SUB R10, R1, R7
30 SUB R11, R1, R8
34 ...
...
64 ADD R12, R3, R4
IM
B
RF
DM
RF
IM
AND
RF
R1
R3
DM
RF
IM
ORR
RF
DM
RF
Flush these instructions
IM
ADD
RF
R3
R4
+
DM
R12
RF

Figure 7.56 Abstract pipeline diagram illustrating earlier branch decision

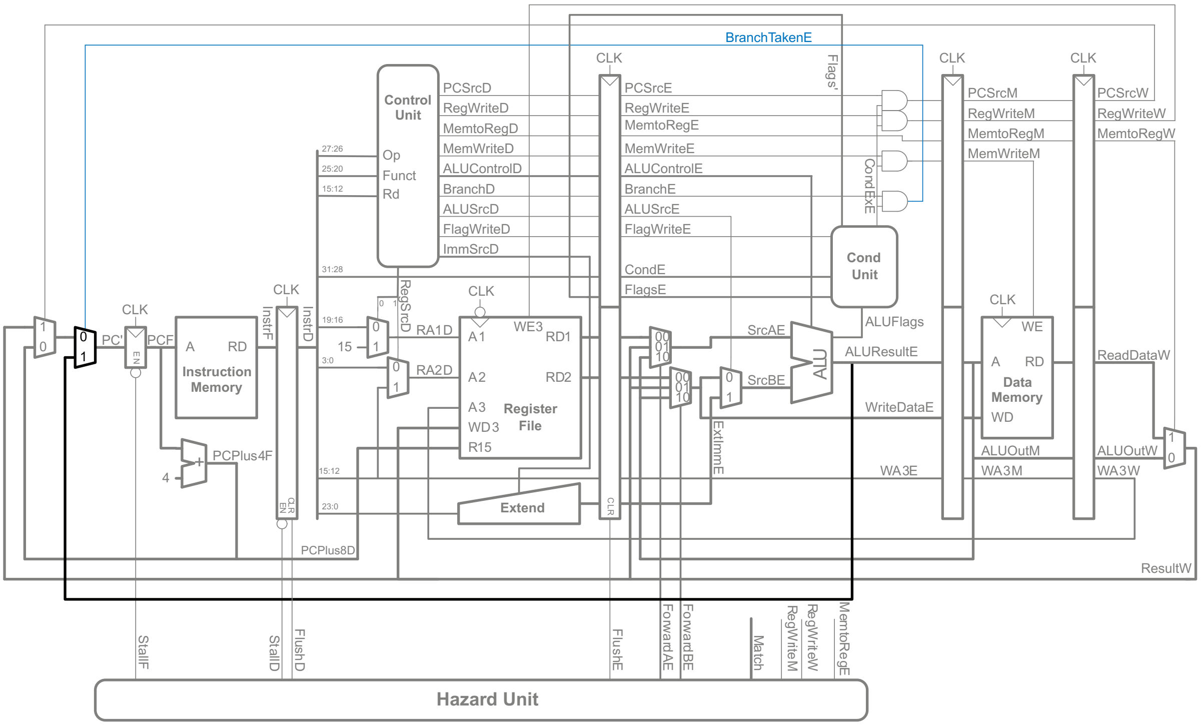

BranchTakenE
Flags'
CLK
CLK
CLK
Control Unit
Op
Funct
Rd
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
BranchD
ALUSrcD
FlagWriteD
ImmSrcD
PCSrcE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
BranchE
ALUSrcE
FlagWriteE
CondE
FlagsE
CondExE
Cond Unit
ALUFlags
PCSrcM
RegWriteM
MemtoRegM
MemWriteM
PCSrcW
RegWriteW
MemtoRegW
27:26
25:20
15:12
31:28
19:16
3:0
15:12
23:0
RegSrcD
CLK
CLK
CLK
PC'
PCF
EN
A RD
Instruction Memory
InstrF
InstrD
PCPlus4F
4
+
15
RA1D
RA2D
WE3
A1
A2
A3
WD3
R15
RD1
RD2
Register File
SrcAE
SrcBE
ALU
ALUResultE
WriteDataE
WA3E
ExtImmE
Extend
CLR
EN
CLR
PCPlus8D
WE
A RD
Data Memory
WD
ALUOutM
WA3M
ReadDataW
ALUOutW
WA3W
ResultW
StallF
StallD
FlushD
FlushE
ForwardAE
ForwardBE
Match
RegWriteM
RegWriteW
MemtoRegE
Hazard Unit

Figure 7.57 Pipelined processor handling branch control hazard

Finally, we must work out the stall and flush signals to handle branches and PC writes. It is common to goof this part of a pipelined processor design because the conditions are rather complicated. When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. When a write to the PC is in the pipeline, the pipeline should be stalled until the write completes. This is done by stalling the Fetch stage. Recall that stalling one stage also requires flushing the next to prevent the instruction from being executed repeatedly. The logic to handle these cases is given here. *PCWrPending* is asserted when a PC write is in progress (in the Decode, Execute, or Memory stage). During this time, the Fetch stage is stalled and the Decode stage is flushed. When the PC write reaches the Writeback stage (*PCSrcW* asserted), *StallF* is released to allow the write to occur, but *FlushD* is still asserted so that the undesired instruction in the Fetch stage does not advance.

To reduce clutter, the Hazard Unit connections of *PCSrcD*, *PCSrcE*, *PCSrcM*, and *BranchTakenE* from the datapath are not shown in Figures 7.57 and 7.58.

```
PCWrPendingF = PCSrcD + PCSrcE + PCSrcM;
StallD = LDRstall;
StallF = LDRstall + PCWrPendingF;
FlushE = LDRstall + BranchTakenE;
FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
```

Branches are very common, and even a two-cycle misprediction penalty still impacts performance. With a bit more work, the penalty could be reduced to one cycle for many branches. The destination address must be computed in the Decode stage as *PCBranchD* = *PCPlus8D* + *ExtImmD*. *BranchTakenD* must also be computed in the Decode stage based on *ALUFlagsE* generated by the previous instruction. This might increase the cycle time of the processor if these flags arrive late. These changes are left as an exercise to the reader (see Exercise 7.36).

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on the result of another instruction that has not yet been written into the register file. The data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong or by stalling the pipeline until the decision is made. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed

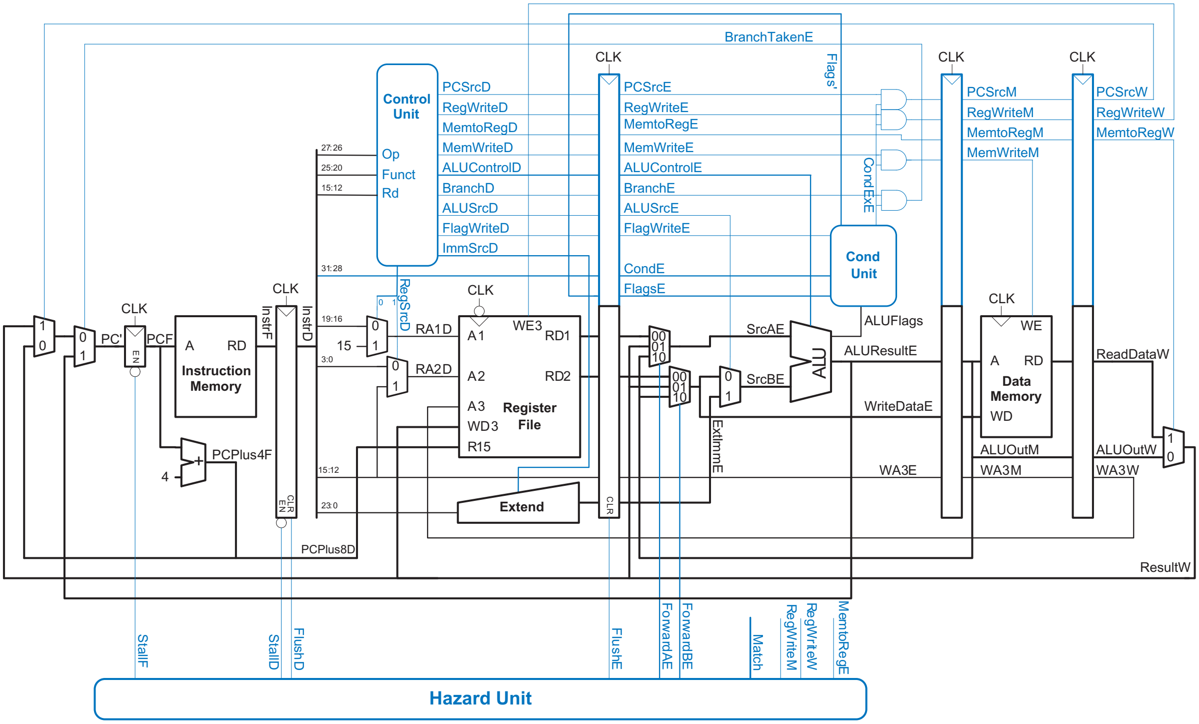

BranchTakenE
CLK
Flags'
CLK
CLK
Control Unit
Op
Funct
Rd
27:26
25:20
15:12
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
BranchD
ALUSrcD
FlagWriteD
ImmSrcD
PCSrcE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
BranchE
ALUSrcE
FlagWriteE
CondE
FlagsE
CondExE
Cond Unit
PCSrcM
RegWriteM
MemtoRegM
MemWriteM
PCSrcW
RegWriteW
MemtoRegW
31:28
RegSrcD
CLK
CLK
CLK
InstrF
InstrD
19:16
15
3:0
RA1D
RA2D
WE3
A1
RD1
A2
RD2
A3
WD3
R15
Register File
SrcAE
SrcBE
ALU
ALUFlags
ALUResultE
WriteDataE
WE
A
RD
Data Memory
WD
ReadDataW
PC'
PCF
A
RD
Instruction Memory
EN
CLR EN
CLR
PCPlus4F
4
15:12
23:0
Extend
ExtImmE
WA3E
ALUOutM
WA3M
ALUOutW
WA3W
PCPlus8D
ResultW
00
01
10
StallF
StallD
FlushD
FlushE
ForwardAE
ForwardBE
Match
RegWriteM
RegWriteW
MemtoRegE
Hazard Unit

Figure 7.58 Pipelined processor with full hazard handling

by now that one of the challenges of designing a pipelined processor is to understand all the possible interactions between instructions and to discover all the hazards that may exist. Figure 7.58 shows the complete pipelined processor handling all of the hazards.

7.5.4 Performance Analysis

The pipelined processor ideally would have a CPI of 1, because a new instruction is issued every cycle. However, a stall or a flush wastes a cycle, so the CPI is slightly higher and depends on the specific program being executed.

Example 7.7 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.5 consists of approximately 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that 50% of the branches are taken (mispredicted), requiring a flush. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of $(0.6)(1) + (0.4)(2) = 1.4$. Branches take one clock cycle when they are predicted properly and three when they are not, so they have a CPI of $(0.5)(1) + (0.5)(3) = 2.0$. All other instructions have a CPI of 1. Hence, for this benchmark, average CPI $= (0.25)(1.4) + (0.1)(1) + (0.13)(2.0) + (0.52)(1) = 1.23$.

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.58. Recall that the register file is written in the first half of the Writeback cycle and read in the second half of the Decode cycle. Therefore, the cycle time of the Decode and Writeback stages is twice the time necessary to do the half-cycle of work.

$$T_{c3} = max\begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & Fetch \\ 2(t_{RFread} + t_{setup}) & Decode \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & Execute \\ t_{pcq} + t_{mem} + t_{setup} & Memory \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & Writeback \end{bmatrix} \tag{7.5}$$

Example 7.8 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance with that of the single-cycle and multicycle processors considered in Example 7.6. The logic delays were given in Table 7.5. Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution: According to Equation 7.5, the cycle time of the pipelined processor is $T_{c3} = \max[40 + 200 + 50,\ 2(100 + 50),\ 40 + 2(25) + 120 + 50,\ 40 + 200 + 50,\ 2(40 + 25 + 60)] = 300$ ps. According to Equation 7.1, the total execution time is $T_3 = (100 \times 10^9$ instructions$)(1.23$ cycles/instruction$)(300 \times 10^{-12}$ s/cycle$) =$ 36.9 seconds. This compares with 84 seconds for the single-cycle processor and 140 seconds for the multicycle processor.

The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the five-fold speed-up one might hope to get from a five-stage pipeline. The pipeline hazards introduce a small CPI penalty. More significantly, the sequencing overhead (clk-to-Q and setup times) of the registers applies to every pipeline stage, not just once to the overall datapath. Sequencing overhead limits the benefits one can hope to achieve from pipelining. The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds eight 32-bit pipeline registers, along with multiplexers, smaller pipeline registers, and control logic to resolve hazards.

7.6 HDL REPRESENTATION*

This section presents HDL code for the single-cycle processor supporting the instructions discussed in this chapter. The code illustrates good coding practices for a moderately complex system. HDL code for the multicycle processor and pipelined processor are left to Exercises 7.25 and 7.40.

In this section, the instruction and data memories are separated from the datapath and connected by address and data busses. In practice, most processors pull instructions and data from separate caches. However, to handle literal pools, a more complete processor must also be able to read data from the instruction memory. Chapter 8 will revisit memory systems, including the interaction of the caches with main memory.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the Decoder and the Conditional Logic. Figure 7.59 shows a block diagram of the single-cycle processor interfaced to external memories.

The HDL code is partitioned into several sections. Section 7.6.1 provides HDL for the single-cycle processor datapath and controller. Section 7.6.2 presents the generic building blocks, such as registers and multiplexers, which are used by any microarchitecture. Section 7.6.3 introduces the testbench and external memories. The HDL is available in electronic form on this book's website (see the Preface).

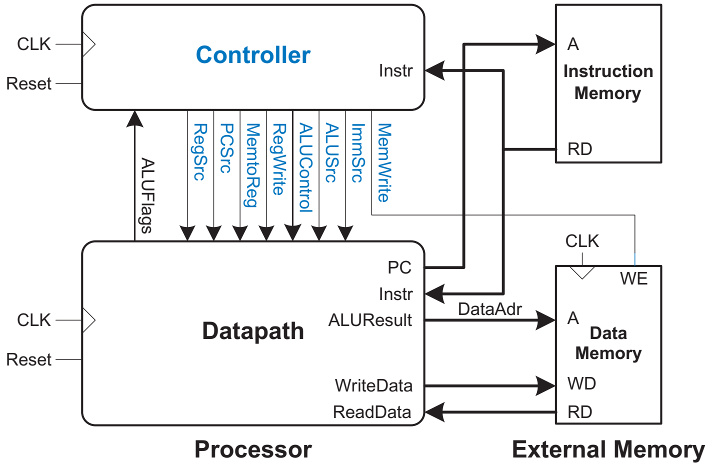

CLK
Reset
Controller
Instr
A
Instruction Memory
RD
ALUFlags
RegSrc
PCSrc
MemtoReg
RegWrite
ALUControl
ALUSrc
ImmSrc
MemWrite
CLK
WE
PC
Instr
ALUResult
DataAdr
CLK
Reset
Datapath
A
Data Memory
WriteData
WD
ReadData
RD
Processor
External Memory

Figure 7.59 **Single-cycle processor interfaced to external memory**

7.6.1 Single-Cycle Processor

The main modules of the single-cycle processor module are given in the following HDL examples.

HDL Example 7.1 SINGLE-CYCLE PROCESSOR

SystemVerilog

```
module arm(input  logic        clk, reset,
           output logic [31:0] PC,
           input  logic [31:0] Instr,
           output logic        MemWrite,
           output logic [31:0] ALUResult, WriteData,
           input  logic [31:0] ReadData);

  logic [3:0] ALUFlags;
  logic       RegWrite,
              ALUSrc, MemtoReg, PCSrc;
  logic [1:0] RegSrc, ImmSrc, ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);
  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- single cycle processor
  port(clk, reset:          in  STD_LOGIC;
       PC:                  out STD_LOGIC_VECTOR(31 downto 0);
       Instr:               in  STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:            out STD_LOGIC;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:            in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
  port(clk, reset:    in  STD_LOGIC;
       Instr:         in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:      in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:        out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:      out STD_LOGIC;
       ImmSrc:        out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:        out STD_LOGIC;
       ALUControl:    out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:      out STD_LOGIC;
       MemtoReg:      out STD_LOGIC;
       PCSrc:         out STD_LOGIC);
  end component;
  component datapath
    port(clk, reset:         in     STD_LOGIC;
      RegSrc:                in     STD_LOGIC_VECTOR(1 downto 0);
      RegWrite:              in     STD_LOGIC;
      ImmSrc:                in     STD_LOGIC_VECTOR(1 downto 0);
      ALUSrc:                in     STD_LOGIC;
      ALUControl:            in     STD_LOGIC_VECTOR(1 downto 0);
      MemtoReg:              in     STD_LOGIC;
      PCSrc:                 in     STD_LOGIC;
      ALUFlags:              out    STD_LOGIC_VECTOR(3 downto 0);
      PC:                    buffer STD_LOGIC_VECTOR(31 downto 0);
      Instr:                 in     STD_LOGIC_VECTOR(31 downto 0);
      ALUResult, WriteData:buffer STD_LOGIC_VECTOR(31 downto 0);
      ReadData:              in     STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
    signal RegSrc, ImmSrc, ALUControl: STD_LOGIC_VECTOR
                                       (1 downto 0);
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
begin
  cont: controller port map(clk, reset, Instr(31 downto 12),
                            ALUFlags, RegSrc, RegWrite,
                            ImmSrc, ALUSrc, ALUControl,
                            MemWrite, MemtoReg, PCSrc);
  dp: datapath port map(clk, reset, RegSrc, RegWrite, ImmSrc,
                        ALUSrc, ALUControl, MemtoReg, PCSrc,
                        ALUFlags, PC, Instr, ALUResult,
                        WriteData, ReadData);
end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic         clk, reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic [1:0]   RegSrc,
                  output logic         RegWrite,
                  output logic [1:0]   ImmSrc,
                  output logic         ALUSrc,
                  output logic [1:0]   ALUControl,
                  output logic         MemWrite, MemtoReg,
                  output logic         PCSrc);
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              FlagW, PCS, RegW, MemW,
              MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control
  port(clk, reset:     in  STD_LOGIC;
       Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:       out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:         out STD_LOGIC;
       ALUControl:     out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       MemtoReg:       out STD_LOGIC;
       PCSrc:          out STD_LOGIC);
end;

architecture struct of controller is
  component decoder
    port(Op:                in  STD_LOGIC_VECTOR(1 downto 0);
         Funct:             in  STD_LOGIC_VECTOR(5 downto 0);
         Rd:                in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:             out STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:   out STD_LOGIC;
         MemtoReg, ALUSrc:  out STD_LOGIC;
         ImmSrc, RegSrc:    out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:        out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component condlogic
    port(clk, reset:       in  STD_LOGIC;
         Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
         ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:  in  STD_LOGIC;
         PCSrc, RegWrite:  out STD_LOGIC;
         MemWrite:         out STD_LOGIC);
  end component;
  signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
  signal PCS, RegW, MemW: STD_LOGIC;
begin
  dec: decoder port map(Instr(27 downto 26), Instr(25 downto 20),
                        Instr(15 downto 12), FlagW, PCS,
                        RegW, MemW, MemtoReg, ALUSrc, ImmSrc,
                        RegSrc, ALUControl);
  cl: condlogic port map(clk, reset, Instr(31 downto 28),
                          ALUFlags, FlagW, PCS, RegW, MemW,
                          PCSrc, RegWrite, MemWrite);
end;
```

HDL Example 7.3 DECODER

SystemVerilog

```
module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc, ALUControl);

  logic [9:0] controls;
  logic       Branch, ALUOp;

  // Main Decoder
  always_comb
      casex(Op)
                                 // Data-processing immediate
        2'b00: if (Funct[5])     controls = 10'b0000101001;
                                 // Data-processing register
               else              controls = 10'b0000001001;
                                 // LDR
        2'b01: if (Funct[0])     controls = 10'b0001111000;
                                 // STR
               else              controls = 10'b1001110100;
                                 // B
        2'b10:                   controls = 10'b0110100010;
                                 // Unimplemented
        default:                 controls = 10'bx;
      endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
    RegW, MemW, Branch, ALUOp} = controls;

  // ALU Decoder
  always_comb
  if (ALUOp) begin            // which DP Instr?
    case(Funct[4:1])
        4'b0100: ALUControl = 2'b00; // ADD
        4'b0010: ALUControl = 2'b01; // SUB
        4'b0000: ALUControl = 2'b10; // AND
        4'b1100: ALUControl = 2'b11; // ORR
        default: ALUControl = 2'bx;  // unimplemented
    endcase

    // update flags if S bit is set (C & V only for arith)
    FlagW[1]    = Funct[0];
    FlagW[0]    = Funct[0] &
      (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
    ALUControl = 2'b00; // add for non-DP instructions
    FlagW      = 2'b00; // don't update Flags
  end

  // PC Logic
  assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
  port(Op:                in  STD_LOGIC_VECTOR(1 downto 0);
       Funct:             in  STD_LOGIC_VECTOR(5 downto 0);
       Rd:                in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:             out STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:   out STD_LOGIC;
       MemtoReg, ALUSrc:  out STD_LOGIC;
       ImmSrc, RegSrc:    out STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:        out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of decoder is
  signal controls:      STD_LOGIC_VECTOR(9 downto 0);
  signal ALUOp, Branch: STD_LOGIC;
  signal op2:           STD_LOGIC_VECTOR(3 downto 0);
begin
  op2 <= (Op, Funct(5), Funct(0));
  process(all) begin -- Main Decoder
    case? (op2) is
      when "000-"  => controls <= "0000001001";
      when "001-"  => controls <= "0000101001";
      when "01-0"  => controls <= "1001110100";
      when "01-1"  => controls <= "0001111000";
      when "10--"  => controls <= "0110100010";
      when others  => controls <= "----------";
    end case?;
  end process;

  (RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
  Branch, ALUOp) <= controls;

  process(all) begin -- ALU Decoder
  if (ALUOp) then
    case Funct(4 downto 1) is
      when "0100"  => ALUControl <= "00"; -- ADD
      when "0010"  => ALUControl <= "01"; -- SUB
      when "0000"  => ALUControl <= "10"; -- AND
      when "1100"  => ALUControl <= "11"; -- ORR
      when others  => ALUControl <= "--"; -- unimplemented
    end case;
    FlagW(1)  <= Funct(0);
    FlagW(0)  <= Funct(0) and (not ALUControl(1));
  else
    ALUControl <= "00";
    FlagW <= "00";
  end if;
  end process;

  PCS <= ((and Rd) and RegW) or Branch;
end;
```

HDL Example 7.4 CONDITIONAL LOGIC

SystemVerilog

```
module condlogic(input  logic        clk, reset,
                 input  logic [3:0]  Cond,
                 input  logic [3:0]  ALUFlags,
                 input  logic [1:0]  FlagW,
                 input  logic        PCS, RegW, MemW,
                 output logic        PCSrc, RegWrite,
                                     MemWrite);

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondEx;
  assign MemWrite  = MemW  & CondEx;
  assign PCSrc     = PCS   & CondEx;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;              // EQ
      4'b0001: CondEx = ~zero;             // NE
      4'b0010: CondEx = carry;             // CS
      4'b0011: CondEx = ~carry;            // CC
      4'b0100: CondEx = neg;               // MI
      4'b0101: CondEx = ~neg;              // PL
      4'b0110: CondEx = overflow;          // VS
      4'b0111: CondEx = ~overflow;         // VC
      4'b1000: CondEx = carry & ~zero;     // HI
      4'b1001: CondEx = ~(carry & ~zero);  // LS
      4'b1010: CondEx = ge;                // GE
      4'b1011: CondEx = ~ge;               // LT
      4'b1100: CondEx = ~zero & ge;        // GT
      4'b1101: CondEx = ~(~zero & ge);     // LE
      4'b1110: CondEx = 1'b1;              // Always
      default: CondEx = 1'bx;              // undefined
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset:       in  STD_LOGIC;
       Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
       ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:  in  STD_LOGIC;
       PCSrc, RegWrite:  out STD_LOGIC;
       MemWrite:         out STD_LOGIC);
end;

architecture behave of condlogic is
  component condcheck
  port(Cond:      in  STD_LOGIC_VECTOR(3 downto 0);
       Flags:     in  STD_LOGIC_VECTOR(3 downto 0);
       CondEx:    out STD_LOGIC);
  end component;
  component flopenr generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:     in  STD_LOGIC_VECTOR (width-1 downto 0);
       q:     out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;
  signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
  signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
  signal CondEx:    STD_LOGIC;
begin
  flagreg1: flopenr generic map(2)
    port map(clk, reset, FlagWrite(1),
             ALUFlags(3 downto 2), Flags(3 downto 2));
  flagreg0: flopenr generic map(2)
    port map(clk, reset, FlagWrite(0),
             ALUFlags(1 downto 0), Flags(1 downto 0));
  cc: condcheck port map(Cond, Flags, CondEx);

  FlagWrite  <=  FlagW and (CondEx, CondEx);
  RegWrite   <=  RegW  and CondEx;
  MemWrite   <=  MemW  and CondEx;
  PCSrc      <=  PCS   and CondEx;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
  port(Cond:     in  STD_LOGIC_VECTOR(3 downto 0);
    Flags:       in  STD_LOGIC_VECTOR(3 downto 0);
    CondEx:      out STD_LOGIC);
end;

architecture behave of condcheck is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);

  process(all) begin -- Condition checking
    case Cond is
      when "0000"  =>  CondEx  <=  zero;
      when "0001"  =>  CondEx  <=  not zero;
      when "0010"  =>  CondEx  <=  carry;
      when "0011"  =>  CondEx  <=  not carry;
      when "0100"  =>  CondEx  <=  neg;
      when "0101"  =>  CondEx  <=  not neg;
      when "0110"  =>  CondEx  <=  overflow;
```

```
        when "0111"  =>  CondEx  <=  not overflow;
        when "1000"  =>  CondEx  <=  carry and (not zero);
        when "1001"  =>  CondEx  <=  not(carry and (not zero));
        when "1010"  =>  CondEx  <=  ge;
        when "1011"  =>  CondEx  <=  not ge;
        when "1100"  =>  CondEx  <=  (not zero) and ge;
        when "1101"  =>  CondEx  <=  not ((not zero) and ge);
        when "1110"  =>  CondEx  <=  '1';
        when others  =>  CondEx  <=  '-';
      end case;
    end process;
  end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic         clk, reset,
                input  logic [1:0]   RegSrc,
                input  logic         RegWrite,
                input  logic [1:0]   ImmSrc,
                input  logic         ALUSrc,
                input  logic [1:0]   ALUControl,
                input  logic         MemtoReg,
                input  logic         PCSrc,
                output logic [3:0]   ALUFlags,
                output logic [31:0]  PC,
                input  logic [31:0]  Instr,
                output logic [31:0]  ALUResult, WriteData,
                input  logic [31:0]  ReadData);

  logic  [31:0]  PCNext, PCPlus4, PCPlus8;
  logic  [31:0]  ExtImm, SrcA, SrcB, Result;
  logic  [3:0]   RA1, RA2;

  // next PC logic
  mux2 #(32)   pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32)  pcreg(clk, reset, PCNext, PC);
  adder #(32)  pcadd1(PC, 32'b100, PCPlus4);
  adder #(32)  pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)  ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile    rf(clk, RegWrite, RA1, RA2,
                Instr[15:12], Result, PCPlus8,
                SrcA, WriteData);
  mux2 #(32) resmux(ALUResult, ReadData, MemtoReg, Result);
  extend     ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:         in     STD_LOGIC;
     RegSrc:               in     STD_LOGIC_VECTOR(1 downto 0);
     RegWrite:             in     STD_LOGIC;
     ImmSrc:               in     STD_LOGIC_VECTOR(1 downto 0);
     ALUSrc:               in     STD_LOGIC;
     ALUControl:           in     STD_LOGIC_VECTOR(1 downto 0);
     MemtoReg:             in     STD_LOGIC;
     PCSrc:                in     STD_LOGIC;
     ALUFlags:             out    STD_LOGIC_VECTOR(3 downto 0);
     PC:                   buffer STD_LOGIC_VECTOR(31 downto 0);
     Instr:                in     STD_LOGIC_VECTOR(31 downto 0);
     ALUResult, WriteData: buffer STD_LOGIC_VECTOR(31 downto 0);
     ReadData:             in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
  port(a, b:         in     STD_LOGIC_VECTOR(31 downto 0);
       ALUControl:   in     STD_LOGIC_VECTOR(1 downto 0);
       Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
       ALUFlags:     out    STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
  port(clk:            in  STD_LOGIC;
       we3:            in  STD_LOGIC;
       ra1, ra2, wa3:  in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:       in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
  port(a, b:  in  STD_LOGIC_VECTOR(31 downto 0);
       y:     out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal PCNext, PCPlus4,
         PCPlus8:        STD_LOGIC_VECTOR(31 downto 0);
  signal ExtImm, Result: STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB:     STD_LOGIC_VECTOR(31 downto 0);
  signal RA1, RA2:       STD_LOGIC_VECTOR(3 downto 0);
begin
  -- next PC logic
  pcmux: mux2 generic map(32)
           port map(PCPlus4, Result, PCSrc, PCNext);
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd1: adder port map(PC, X"00000004", PCPlus4);
  pcadd2: adder port map(PCPlus4, X"00000004", PCPlus8);

  -- register file logic
  ra1mux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
                       Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
                     Instr(15 downto 12), Result,
                     PCPlus8, SrcA, WriteData);
  resmux: mux2 generic map(32)
    port map(ALUResult, ReadData, MemtoReg, Result);
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic
  srcbmux: mux2 generic map(32)
    port map(WriteData, ExtImm, ALUSrc, SrcB);
  i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult,
                     ALUFlags);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any digital system, including a register file, adder, flip-flops, and a 2:1 multiplexer. The HDL for the ALU is left to Exercises 5.11 and 5.12.

HDL Example 7.6 REGISTER FILE

SystemVerilog

```
module regfile(input  logic         clk,
               input  logic         we3,
               input  logic  [3:0]  ra1, ra2, wa3,
               input  logic  [31:0] wd3, r15,
               output logic  [31:0] rd1, rd2);

  logic [31:0] rf[14:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 15 reads PC+8 instead

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
  if rising_edge(clk) then
    if we3 = '1' then mem(to_integer(wa3)) <= wd3;
    end if;
  end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

HDL Example 7.7 ADDER

SystemVerilog

```
module adder #(parameter WIDTH = 8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL Example 7.8 IMMEDIATE EXTENSION

SystemVerilog

```
module extend(input  logic  [23:0]  Instr,
              input  logic  [1:0]   ImmSrc,
              output logic  [31:0]  ExtImm);

  always_comb
    case(ImmSrc)
                // 8-bit unsigned immediate
      2'b00:    ExtImm = {24'b0, Instr[7:0]};
                // 12-bit unsigned immediate
      2'b01:    ExtImm = {20'b0, Instr[11:0]};
                // 24-bit two's complement shifted branch
      2'b10:    ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default:  ExtImm = 32'bx; // undefined
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(all) begin
    case ImmSrc is
      when "00"  => ExtImm <= (X"000000", Instr(7 downto 0));
      when "01"  => ExtImm <= (X"00000", Instr(11 downto 0));
      when "10"  => ExtImm <= (Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23 downto 0), "00");
      when others  =>  ExtImm  <=  X"--------";
    end case;
  end process;
end;
```

HDL Example 7.9 RESETTABLE FLIP-FLOP

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
             (input  logic              clk, reset,
              input  logic  [WIDTH-1:0] d,
              output logic  [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)  q <= 0;
    else        q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is -- flip-flop with synchronous reset
  generic(width: integer);
  port(clk, reset:  in  STD_LOGIC;
       d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:           out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
        q <= d;
    end if;
  end process;
end;
```

HDL Example 7.10 RESETTABLE FLIP-FLOP WITH ENABLE

SystemVerilog

```
module flopenr #(parameter WIDTH = 8)
               (input  logic              clk, reset, en,
                input  logic  [WIDTH-1:0] d,
                output logic  [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is -- flip-flop with enable and synchronous reset
  generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:     in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:     out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.11 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
            (input   logic  [WIDTH-1:0]  d0, d1,
             input   logic               s,
             output  logic  [WIDTH-1:0]  y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1:  in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:       in  STD_LOGIC;
       y:       out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;
```

7.6.3 Testbench

The testbench loads a program into the memories. The program in Figure 7.60 exercises all of the instructions by performing a computation that should produce the correct result only if all of the instructions are functioning correctly. Specifically, the program will write the value 7 to address 100 if it runs correctly, but it is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

The machine code is stored in a hexadecimal file called memfile.dat, which is loaded by the testbench during simulation. The file consists of the machine code for the instructions, one instruction per line. The testbench, top-level ARM module, and external memory HDL code are given in the following examples. The memories in this example hold 64 words each.

```
ADDR         PROGRAM            ; COMMENTS                       BINARY MACHINE CODE                          HEX CODE
00  MAIN     SUB R0, R15, R15   ; R0 = 0                         1110 000 0010 0 1111 0000 0000 0000 1111     E04F000F
04           ADD R2, R0, #5     ; R2 = 5                         1110 001 0100 0 0000 0010 0000 0000 0101     E2802005
08           ADD R3, R0, #12    ; R3 = 12                        1110 001 0100 0 0000 0011 0000 0000 1100     E280300C
0C           SUB R7, R3, #9     ; R7 = 3                         1110 001 0010 0 0011 0111 0000 0000 1001     E2437009
10           ORR R4, R7, R2     ; R4 = 3 OR 5 = 7                1110 000 1100 0 0111 0100 0000 0000 0010     E1874002
14           AND R5, R3, R4     ; R5 = 12 AND 7 = 4              1110 000 0000 0 0011 0101 0000 0000 0100     E0035004
18           ADD R5, R5, R4     ; R5 = 4 + 7 = 11                1110 000 0100 0 0101 0101 0000 0000 0100     E0855004
1C           SUBS R8, R5, R7    ; R8 = 11 - 3 = 8, set Flags     1110 000 0010 1 0101 1000 0000 0000 0111     E0558007
20           BEQ END            ; shouldn't be taken             0000 1010 0000  0000 0000 0000 0000 1100     0A00000C
24           SUBS R8, R3, R4    ; R8 = 12 - 7  = 5               1110 000 0010 1 0011 1000 0000 0000 0100     E0538004
28           BGE AROUND         ; should be taken                1010 1010 0000  0000 0000 0000 0000 0000     AA000000
2C           ADD R5, R0, #0     ; should be ski pped             1110 001 0100 0 0000 0101 0000 0000 0000     E2805000
30 AROUND    SUBS R8, R7, R2    ; R8 = 3 - 5 = -2, set Flags     1110 000 0010 1 0111 1000 0000 0000 0010     E0578002
34           ADDLT R7, R5, #1   ; R7 = 11 + 1 = 12               1011 001 0100 0 0101 0111 0000 0000 0001     B2857001
38           SUB R7, R7, R2     ; R7 = 12 - 5 = 7                1110 000 0010 0 0111 0111 0000 0000 0010     E0477002
3C           STR R7, [R3, #84]  ; mem[12+84] = 7                 1110 010 1100 0 0011 0111 0000 0101 0100     E5837054
40           LDR R2, [R0, #96]  ; R2 = mem[96] = 7               1110 010 1100 1 0000 0010 0000 0110 0000     E5902060
44           ADD R15, R15, R0   ; PC = PC+8 (skips next)         1110 000 0100 0 1111 1111 0000 0000 0000     E08FF000
48           ADD R2, R0, #14    ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001     E280200E
4C           B END              ; always taken                   1110 1010 0000  0000 0000 0000 0000 0001     EA000001
50           ADD R2, R0, #13    ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001     E280200D
54           ADD R2, R0, #10    ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001     E280200A
58 END       STR R2, [R0, #100] ; mem[100] = 7                   1110 010 1100 0 0000 0010 0000 0101 0100     E5802064
```

Figure 7.60 Assembly and machine code for test program

HDL Example 7.12 TESTBENCH

SystemVerilog

```
module testbench();
  logic        clk;
  logic        reset;
  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
  begin
    reset <= 1; # 22; reset <= 0;
  end

  // generate clock to sequence tests
  always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:          in  STD_LOGIC;
         WriteData, DatAadr:  out STD_LOGIC_VECTOR(31 downto 0);
         MemWrite:            out STD_LOGIC);
  end component;
  signal WriteData, DataAdr:     STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset,  MemWrite: STD_LOGIC;
begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;
  // check that 7 gets written to address 0x64
  // at end of program
  always @(negedge clk)
  begin
    if(MemWrite) begin
      if(DataAdr === 100 & WriteData === 7) begin
        $display("Simulation succeeded");
        $stop;
      end else if (DataAdr !== 96) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

```
  -- generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 0x64
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and MemWrite = '1') then
      if (to_integer(DataAdr) = 100 and
         to_integer(WriteData) = 7) then
        report "NO ERRORS: Simulation succeeded" severity
        failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
          WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
  port(clk, reset:         in     STD_LOGIC;
       WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:           buffer STD_LOGIC);
end;

architecture test of top is
  component arm
    port(clk, reset:        in  STD_LOGIC;
         PC:                out STD_LOGIC_VECTOR(31 downto 0);
         Instr:             in  STD_LOGIC_VECTOR(31 downto 0);
         MemWrite:          out STD_LOGIC;
         ALUResult, WriteData:out STD_LOGIC_VECTOR(31 downto 0);
         ReadData:          in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
  port(clk, we:  in   STD_LOGIC;
    a, wd:       in   STD_LOGIC_VECTOR(31 downto 0);
    rd:          out  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal PC, Instr,
         ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  i_arm: arm port map(clk, reset, PC, Instr, MemWrite, DataAdr,
                      WriteData, ReadData);
  i_imem: imem port map(PC, Instr);
  i_dmem: dmem port map(clk, MemWrite, DataAdr,
                        WriteData, ReadData);
end;
```

HDL Example 7.14 DATA MEMORY

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
  port(clk, we:  in  STD_LOGIC;
       a, wd:    in  STD_LOGIC_VECTOR(31 downto 0);
       rd:       out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                  STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin -- read or write memory
    loop
      if clk'event and clk = '1' then
        if (we = '1') then
          mem(to_integer(a(7 downto 2))) := wd;
        end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

HDL Example 7.15 INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
  port(a:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:  out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
                       STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
```

```
        index := 0;
        FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
        while not endfile(mem_file) loop
          readline(mem_file, L);
          result := 0;
          for i in 1 to 8 loop
            read(L, ch);
            if '0' <= ch and ch <= '9' then
              result := character'pos(ch) - character'pos('0');
            elsif 'a' <= ch and ch <= 'f' then
              result := character'pos(ch) - character'pos('a') + 10;
            elsif 'A' <= ch and ch <= 'F' then
              result := character'pos(ch) - character'pos('A') + 10;
            else report "Format error on line " & integer'image(index)
              severity error;
            end if;
            mem(index)(35-i*4 downto 32-i*4) :=
              to_std_logic_vector(result,4);
          end loop;
          index := index + 1;
        end loop;

        -- read memory
        loop
          rd <= mem(to_integer(a(7 downto 2)));
          wait on a;
        end loop;
      end process;
    end;
```

7.7 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance, we would like to speed-up the clock and/or reduce the CPI. This section surveys some existing speed-up techniques. The implementation details become quite complex, so we focus on the concepts. Hennessy & Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Advances in integrated circuit manufacturing have steadily reduced transistor sizes. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption is now an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.7.1 Deep Pipelines

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 10–20 stages are now commonly used.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

In the late 1990s and early 2000s, microprocessors were marketed largely based on clock frequency ($1/T_c$). This pushed microprocessors to use very deep pipelines (20–31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits for overall performance were questionable. Power is proportional to clock frequency and also increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are decreasing.

Example 7.9

Consider building a pipelined processor by chopping up the single-cycle processor into *N* stages. The single-cycle processor has a propagation delay of 740 ps through the combinational logic. The sequencing overhead of a register is 90 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.7 has a CPI of 1.23. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution: The cycle time for an *N*-stage pipeline is $T_c = (740/N + 90)$ ps. The CPI is $1.23 + 0.1(N–5)$. The time per instruction, or instruction time, is the product of the cycle time and the CPI. Figure 7.61 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 279 ps at $N = 8$ stages. This minimum is only slightly better than the 293 ps per instruction achieved with a five-stage pipeline.

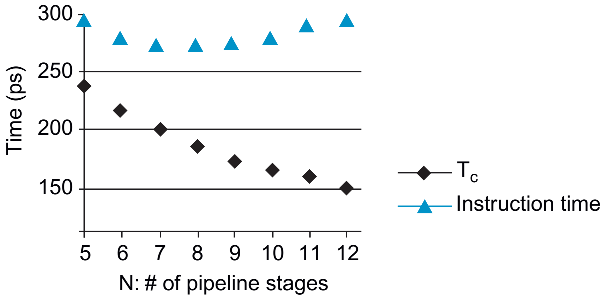


Figure 7.61 **Cycle time and instruction time vs. the number of pipeline stages**

7.7.2 Micro-Operations

Recall our design principles of “regularity supports simplicity” and “make the common case fast.” Pure reduced instruction set computer (RISC) architectures such as MIPS contain only simple instructions, typically those that can be executed in a single cycle on a simple, fast datapath with a three-ported register file, single ALU, and single data memory access like the ones we have developed in this chapter. Complex instruction set computer (CISC) architectures generally include instructions requiring more registers, more additions, or more than one memory access per instruction. For example, the x86 instruction `ADD [ESP], [EDX + 80 + EDI*2]` involves reading the three registers, adding the base, displacement, and scaled index, reading two memory locations, summing their values, and writing the result back to memory. A microprocessor that could perform all of these functions at once would be unnecessarily slow on more common, simpler instructions.

Computer architects make the common case fast by defining a set of simple *micro-operations* (also known as *micro-ops* or *μops*) that can be executed on simple datapaths. Each real instruction is decoded into one or more micro-ops. For example, if we defined *μops* resembling basic ARM instructions and some temporary registers T1 and T2 for holding intermediate results, then the x86 instruction could become seven *μops*:

```
ADD  T1, [EDX + 80] ; T1 <— EDX + 80
LSL  T2, EDI, 2     ; T2 <— EDI*2
ADD  T1, T2, T2     ; T1 <— EDX + 80 + EDI*2
LDR  T1, [T1]       ; T1 <— MEM[EDX + 80 + EDI*2]
LDR  T2, [ESP]      ; T2 <— MEM[ESP]
ADD  T1, T2, T1     ; T1 <— MEM[ESP] + MEM[EDX + 80 + EDI*2]
STR  T1, [ESP]      ; MEM[ESP]<— MEM[ESP] + MEM[EDX + 80 + EDI*2]
```

Although most ARM instructions are simple, some are decomposed into multiple micro-ops as well. For example, loads with postindexed addressing (such as `LDR R1, [R2], #4`) require a second write port on the register file. Data-processing instructions with register-shifted register addressing (such as `ORR R3, R4, R5, LSL R6`) require a third read port on the register file. Instead of providing a larger five-port register file, the ARM datapath may decode these complex instructions into pairs of simpler instructions:

Complex Op	**Micro-op Sequence**
`LDR R1, [R2], #4`	`LDR R1, [R2]` `ADD R2, R2, #4`
`ORR R3, R4, R5 LSL R6`	`LSL T1, R5, R6` `ORR R3, R4, T1`

Although the programmer could have written the simpler instructions directly and the program may have run just as fast, a single complex instruction takes less memory than the pair of simpler instructions. Reading instructions from external memory can consume significant power, so the complex instruction also can save power. The ARM instruction set is so successful in part because of the architects' judicious choice of instructions that give better code density than pure RISC instructions sets such as MIPS, yet more efficient decoding than CISC instruction sets such as x86.

Microarchitects make the decision of whether to provide hardware to implement a complex operation directly or break it into micro-op sequences. They make similar decisions about other options described later in this section. These choices lead to different points in the performance-power-cost design space.

7.7.3 Branch Prediction

An ideal pipelined processor would have a CPI of 1.0. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a for or while loop). Loops tend to be executed many times, so these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken. This is called *static branch prediction*, because it does not depend on the history of the program.

Forward branches are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

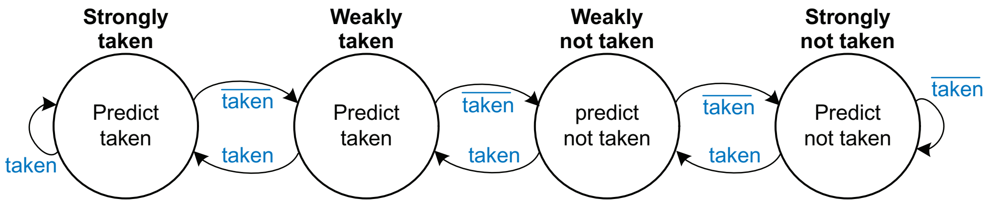

Strongly taken
Weakly taken
Weakly not taken
Strongly not taken
Predict taken
Predict taken
predict not taken
Predict not taken
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$

Figure 7.62 Two-bit branch predictor state transition diagram

To see the operation of dynamic branch predictors, consider the following loop from Code Example 6.17. The loop repeats 10 times, and the `BGE` out of the loop is taken only on the last iteration.

```
  MOV  R1,  #0
  MOV  R0,  #0
FOR
  CMP  R0,  #10
  BGE  DONE
  ADD  R1,  R1,  R0
  ADD  R0,  R0,  #1
  B    FOR
DONE
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `BGE` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A *two-bit dynamic branch predictor* solves this problem by having four states: *strongly taken*, *weakly taken*, *weakly not taken*, and *strongly not taken*, as shown in Figure 7.62. When the loop is repeating, it enters the "strongly not taken" state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the "weakly not taken" state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and re-enters the "strongly not taken" state. In summary, a two-bit branch predictor mispredicts only the last branch of a loop.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

7.7.4 Superscalar Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.63 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

Figure 7.64 shows a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or IPC. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.65 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are shown in blue. The `ADD` instruction is dependent on R8, which is produced by the `LDR` instruction, so it cannot be issued at the same time as `LDR`. The `ADD` instruction stalls for yet another cycle so that `LDR` can forward R8 to `ADD` in cycle 5. The other dependencies (between `SUB` and

A *scalar* processor acts on one piece of data at a time.
A *vector* processor acts on several pieces of data with a single instruction.
A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Our ARM pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980s and 1990s because they efficiently handled the long vectors of data common in scientific computations, and they are heavily used now in *graphics processing units* (GPUs). Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

However, modern processors also include hardware to handle short vectors of data that are common in multimedia and graphics applications. These are called *single instruction multiple data* (SIMD) units and are discussed in Section 6.7.5.

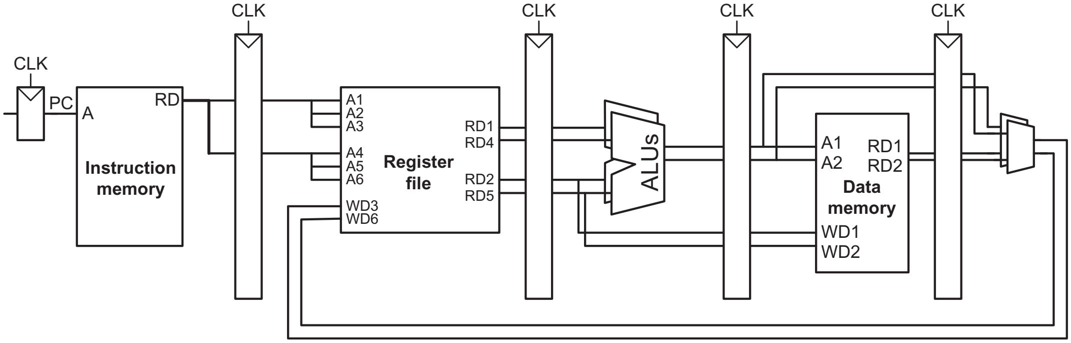


Figure 7.63 Superscalar datapath

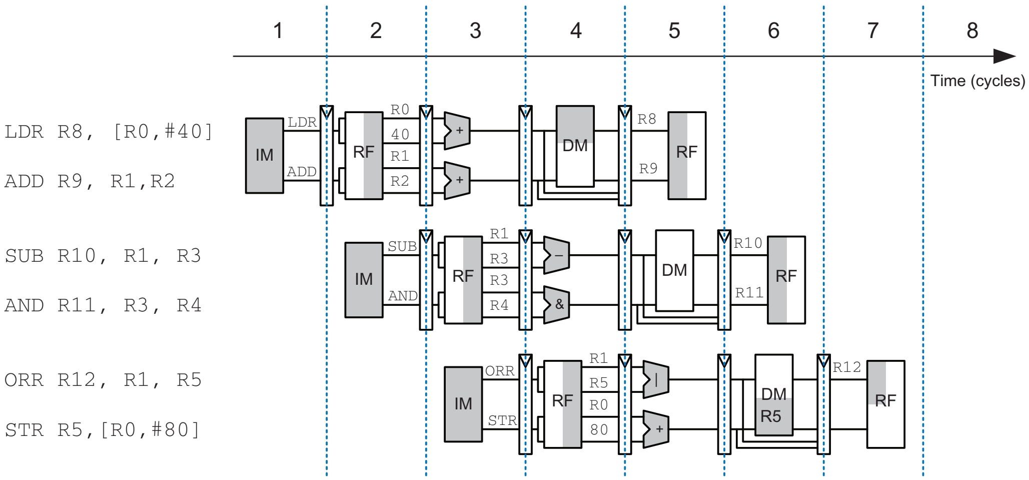

1
2
3
4
5
6
7
8
Time (cycles)
LDR R8, [R0,#40]
ADD R9, R1,R2
SUB R10, R1, R3
AND R11, R3, R4
ORR R12, R1, R5
STR R5,[R0,#80]
IM
LDR
ADD
RF
R0
40
R1
R2
+
DM
R8
R9
RF
SUB
AND
R3
R4
−
&
R10
R11
ORR
STR
R5
80
|
R5
R12

Figure 7.64 Abstract view of a superscalar pipeline in operation

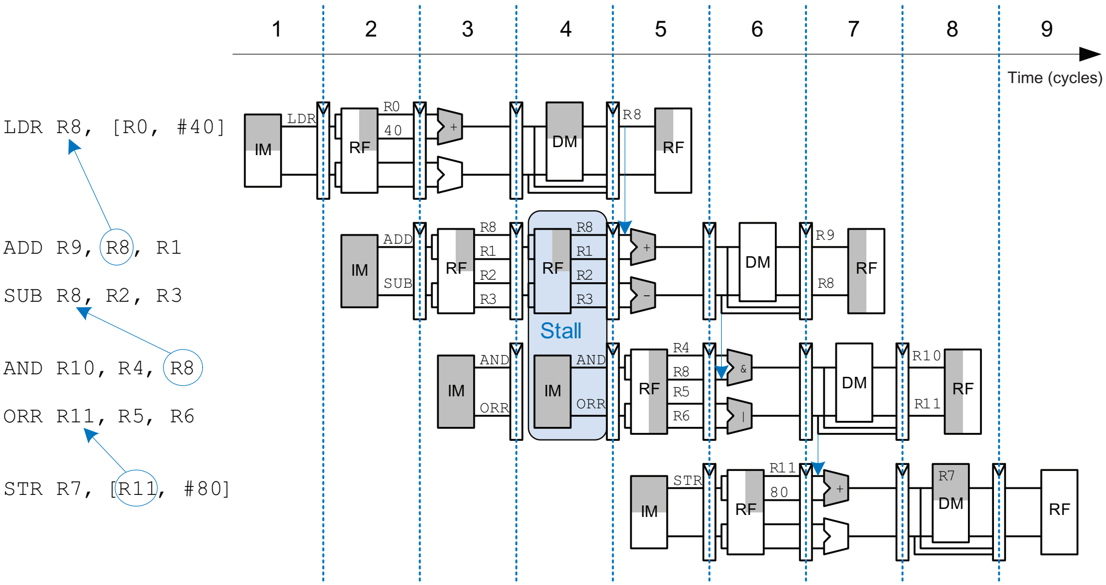

1
2
3
4
5
6
7
8
9
Time (cycles)
LDR R8, [R0, #40]
ADD R9, R8, R1
SUB R8, R2, R3
AND R10, R4, R8
ORR R11, R5, R6
STR R7, [R11, #80]
IM
LDR
RF
R0
40
+
DM
R8
RF
ADD
SUB
R1
R2
R3
Stall
R9
AND
ORR
R4
R5
R6
&
|
R10
R11
STR
80
R7

Figure 7.65 Program with data dependencies

AND based on R8, and between ORR and STR based on R11) are handled by forwarding results produced in one cycle to be consumed in the next. This program requires five cycles to issue six instructions, for an IPC of 1.2.

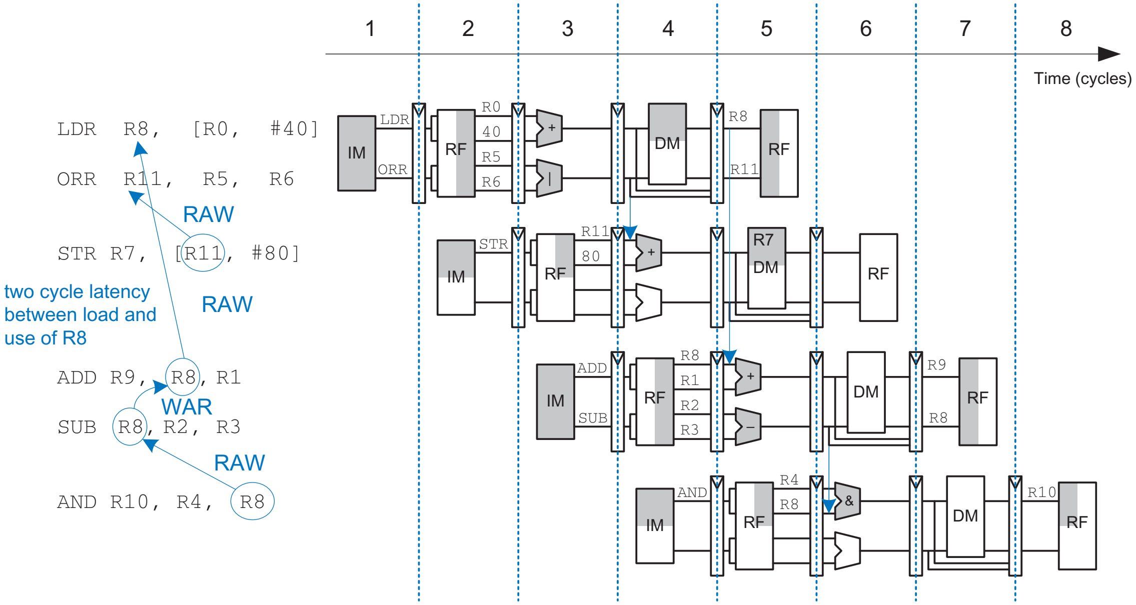
Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

7.7.5 Out-of-Order Processor

To cope with the problem of dependencies, an out-of-order processor looks ahead across many instructions to *issue*, or begin executing, independent instructions as rapidly as possible. The instructions can issue in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.

Consider running the same program from Figure 7.65 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.66 shows the data dependencies and the



```
LDR  R8,  [R0,  #40]
ORR  R11,  R5,  R6
STR R7,  [R11,  #80]
ADD R9, R8, R1
SUB R8, R2, R3
AND R10, R4, R8
```

1
2
3
4
5
6
7
8
Time (cycles)
RAW
two cycle latency between load and use of R8
RAW
WAR
RAW
IM
LDR
ORR
RF
R0
40
R5
R6
+
|
DM
R8
R11
RF
IM
STR
RF
R11
80
+
R7
DM
RF
IM
ADD
SUB
RF
R8
R1
R2
R3
+
−
DM
R9
R8
RF
IM
AND
RF
R4
R8
&
DM
R10
RF

Figure 7.66 Out-of-order execution of a program with dependencies

operation of the processor. The classifications of dependencies as RAW and WAR will be discussed soon. The constraints on issuing instructions are:

- Cycle 1
  - The `LDR` instruction issues.
  - The `ADD`, `SUB`, and `AND` instructions are dependent on `LDR` by way of R8, so they cannot issue yet. However, the `ORR` instruction is independent, so it also issues.
- Cycle 2
  - Remember that there is a two-cycle latency between issuing an `LDR` instruction and a dependent instruction, so `ADD` cannot issue yet because of the R8 dependence. `SUB` writes R8, so it cannot issue before `ADD`, lest `ADD` receive the wrong value of R8. `AND` is dependent on `SUB`.
  - Only the `STR` instruction issues.
- Cycle 3
  - On cycle 3, R8 is available, so the `ADD` issues. `SUB` issues simultaneously, because it will not write R8 until after `ADD` consumes R8.
- Cycle 4
  - The `AND` instruction issues. R8 is forwarded from `SUB` to `AND`.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.

The dependence of `ADD` on `LDR` by way of R8 is a *read after write* (*RAW*) hazard. `ADD` must not read R8 until after `LDR` has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of `STR` on `ORR` by way of R11 and of `AND` on `SUB` by way of R8 are RAW dependencies.

The dependence between `SUB` and `ADD` by way of R8 is called a *write after read* (*WAR*) hazard or an *antidependence*. `SUB` must not write R8 before `ADD` reads R8, so that `ADD` receives the correct value according to the original order of the program. WAR hazards could not occur in the simple pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, `SUB`) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the `SUB` instruction had written R12 instead of R8, then the dependency would disappear and `SUB` could be issued before `ADD`. The ARM architecture only has 16 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all the other registers are in use.

A third type of hazard, not shown in the program, is called a *write after write* (*WAW*) hazard or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being written to the register. For example, in the following code, `LDR` and `ADD` both write R8. The final value in R8 should come from `ADD` according to the order of the program. If an out-of-order processor attempted to execute `ADD` first, then a WAW hazard would occur.

```
LDR  R8,  [R3]
ADD  R8,  R1,  R2
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer's using the same destination register for two unrelated instructions. If the `ADD` instruction were issued first, then the program could eliminate the WAW hazard by discarding the result of the `LDR` instead of writing it to R8. This is called *squashing* the `LDR`.[4]

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction level parallelism* (ILP) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than two or three, even with six-way superscalar datapaths with out-of-order execution.

7.7.6 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR and WAW hazards. Register renaming adds some nonarchitectural renaming registers to the processor. For example, a processor might add 20 renaming registers, called T0–T19. The

---

[4] You might wonder why the `LDR` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `LDR` potentially may produce a Data Abort exception, so it must be issued to check for the exception, even though the result can be discarded.

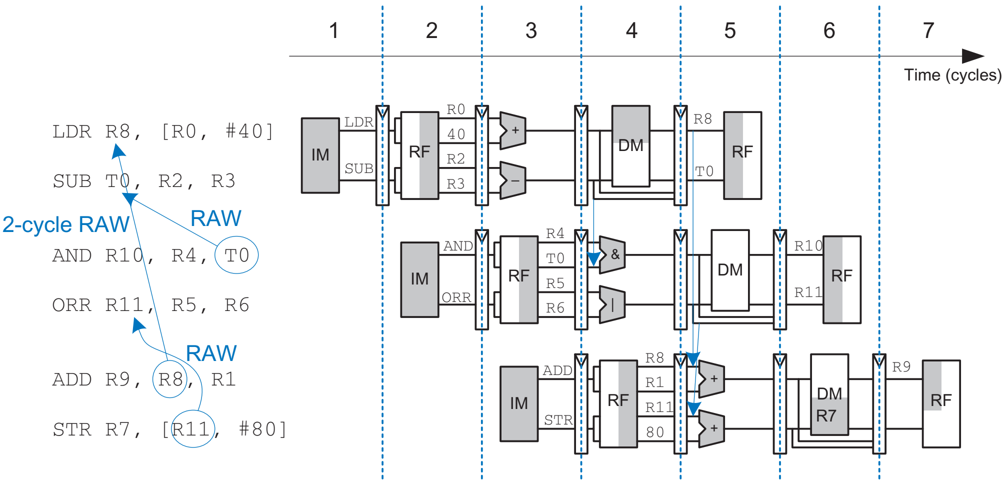

1
2
3
4
5
6
7
Time (cycles)
LDR R8, [R0, #40]
SUB T0, R2, R3
2-cycle RAW
RAW
AND R10, R4, T0
ORR R11, R5, R6
RAW
ADD R9, R8, R1
STR R7, [R11, #80]
IM
LDR
SUB
RF
R0
40
R2
R3
DM
R8
T0
RF
IM
AND
ORR
RF
R4
T0
R5
R6
DM
R10
R11
RF
IM
ADD
STR
RF
R8
R1
R11
80
DM
R7
R9
RF

Figure 7.67 Out-of-order execution of a program using register renaming

programmer cannot use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

For example, in the previous section, a WAR hazard occurred between the `SUB` and `ADD` instructions based on reusing R8. The out-of-order processor could rename R8 to T0 for the `SUB` instruction. Then, `SUB` could be executed sooner, because T0 has no dependency on the `ADD` instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, R8 must also be renamed to T0 in the `AND` instruction, because it refers to the result of `SUB`.

Figure 7.67 shows the same program from Figure 7.65 executing on an out-of-order processor with register renaming. R8 is renamed to T0 in `SUB` and `AND` to eliminate the WAR hazard. The constraints on issuing instructions are:

- Cycle 1
  - The `LDR` instruction issues.
  - The `ADD` instruction is dependent on `LDR` by way of R8, so it cannot issue yet. However, the `SUB` instruction is independent now that its destination has been renamed to T0, so `SUB` also issues.
- Cycle 2
  - Remember that there is a two-cycle latency between issuing an `LDR` instruction and a dependent instruction, so `ADD` cannot issue yet because of the R8 dependence.

  - The `AND` instruction is dependent on `SUB`, so it can issue. T0 is forwarded from `SUB` to `AND`.
  - The `ORR` instruction is independent, so it also issues.
- Cycle 3
  - On cycle 3, R8 is available, so the `ADD` issues.
  - R11 is also available, so `STR` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.7.7 Multithreading

Because the ILP of real programs tends to be fairly low, adding more execution units to a superscalar or out-of-order processor gives diminishing returns. Another problem, discussed in Chapter 8, is that memory is much slower than the processor. Most loads and stores access a smaller and faster memory, called a *cache*. However, when the instructions or data are not available in the cache, the processor may stall for 100 or more cycles while retrieving the information from the main memory. Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

To explain multithreading, we need to define a few new terms. A program running on a computer is called a *process*. Computers can run multiple processes simultaneously; for example, you can play music on a PC while surfing the web and running a virus checker. Each process consists of one or more *threads* that also run simultaneously. For example, a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user works, and a third thread printing the document. In this way, the user does not have to wait, for example, for a document to finish printing before being able to type again. The degree to which a process can be split into multiple threads that can run simultaneously defines its level of *thread level parallelism* (TLP).

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the OS. When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*. As long as the processor switches through all the threads fast enough, the user perceives all of the threads as running at the same time.

A multithreaded processor contains more than one copy of its architectural state, so that more than one thread can be active at a time. For example, if we extended a processor to have four program counters and 64 registers, four threads could be available at one time. If one thread stalls while waiting for data from main memory, then the processor could context switch to another thread without any delay, because the program counter and registers are already available. Moreover, if one thread lacks sufficient parallelism to keep all the execution units busy in a superscalar design, then another thread could issue instructions to the idle units.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread. Multithreading is also relatively inexpensive to implement, because it replicates only the PC and register file, not the execution units and memories.

7.7.8 Multiprocessors

With contributions from Matthew Watkins

Modern processors have enormous numbers of transistors available. Using them to increase the pipeline depth or to add more execution units to a superscalar processor gives little performance benefit and is wasteful of power. Around the year 2005, computer architects made a major shift to building multiple copies of the processor on the same chip; these copies are called *cores*.

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. Three common classes of multiprocessors include *symmetric* (or *homogeneous*) multiprocessors, *heterogeneous* multiprocessors, and *clusters*.

Symmetric Multiprocessors

Symmetric multiprocessors include two or more identical processors sharing a single main memory. The multiple processors may be separate chips or multiple cores on the same chip.

Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running more threads simultaneously is easy; the threads are simply divided up among the processors. Unfortunately, typical PC users need to run only a small number of threads at any given time. Running a particular thread faster is much more challenging. The programmer must divide the existing thread into multiple threads to execute on each processor. This becomes tricky when the processors need to communicate with each other. One of the major challenges for computer designers and programmers is to effectively use large numbers of processor cores.

Symmetric multiprocessors have a number of advantages. They are relatively simple to design because the processor can be designed once and then replicated multiple times to increase performance. Programming for and executing code on a symmetric multiprocessor is also relatively straightforward because any program can run on any processor in the system and achieve approximately the same performance.

Heterogeneous Multiprocessors

Unfortunately, continuing to add more and more symmetric cores is not guaranteed to provide continued performance improvement. As of 2015, consumer applications used few threads at any given time, and a typical consumer might be expected to have a couple of applications actually computing simultaneously. Although this is enough to keep dual-core and quad-core systems busy, unless programs start incorporating significantly more parallelism, continuing to add more cores beyond this point will provide diminishing benefits. As an added issue, because general-purpose processors are designed to provide good average performance, they are generally not the most power-efficient option for performing a given operation. This energy inefficiency is especially important in highly power-constrained systems such as mobile phones.

Heterogeneous multiprocessors aim to address these issues by incorporating different types of cores and/or specialized hardware in a single system. Each application uses those resources that provide the best performance, or power-performance ratio, for that application. Because transistors are fairly plentiful these days, the fact that not every application will make use of every piece of hardware is of lesser concern. Heterogeneous systems can take a number of forms. A heterogeneous system can incorporate cores with different microarchitectures that have different power, performance, and area trade-offs.

One heterogeneous strategy popularized by ARM is *big.LITTLE*, in which a system contains both energy-efficient and high-performance cores. “LITTLE” cores such as the Cortex-A53 are single-issue or dual-issue in-order processors with good energy efficiency that handle routine tasks. “big” cores such as the Cortex-A57 are more complex superscalar out-of-order cores delivering high performance for peak loads.

Another heterogeneous strategy is accelerators, in which a system contains special-purpose hardware optimized for performance or energy efficiency on specific types of tasks. For example, a mobile system-on-chip (SoC) presently may contain dedicated accelerators for graphics processing, video, wireless communication, real-time tasks, and cryptography. These accelerators can be 10–100x more efficient than general-purpose processors for the same tasks. Digital signal processors are another class of accelerators. These processors have a specialized instruction set optimized for math-intensive tasks.

Heterogeneous systems are not without their drawbacks. They add complexity in terms of both designing the different heterogeneous elements and the additional programming effort to decide when and how to make use of the varying resources. Symmetric and heterogeneous

Scientists searching for signs of extraterrestrial intelligence use the world’s largest clustered multiprocessors to analyze radio telescope data for patterns that might be signs of life in other solar systems. The cluster, operational since 1999, consists of personal computers owned by more than 6 million volunteers around the world.

When a computer in the cluster is idle, it fetches a piece of the data from a centralized server, analyzes the data, and sends the results back to the server. You can volunteer your computer’s idle time for the cluster by visiting `setiathome.berkeley.edu`.

systems both have their places in modern systems. Symmetric multiprocessors are good for situations like large data centers that have lots of thread level parallelism available. Heterogeneous systems are good for cases that have more varying or special-purpose workloads.

Clusters

In *clustered* multiprocessors, each processor has its own local memory system. One type of cluster is a group of personal computers connected together on the network running software to jointly solve a large problem. Another type of cluster that has become very important is the *data center*, in which racks of computers and disks are networked together and share power and cooling. Major Internet companies including Google, Amazon, and Facebook have driven the rapid development of data centers to support millions of users around the world.

7.8 REAL-WORLD PERSPECTIVE: EVOLUTION OF ARM MICROARCHITECTURE*

DMIPS (Dhrystone millions of instructions per second) measures performance.

This section traces the development of the ARM architecture and microarchitecture since its inception in 1985. Table 7.7 summarizes the highlights, showing 10x improvement in IPC and 250x increase in

Table 7.7 Evolution of ARM processors

Microarchitecture	Year	Architecture	Pipeline Depth	DMIPS/ MHz	Representative Frequency (MHz)	L1 Cache	Relative Size
ARM1	1985	v1	3	0.33	8	N/A	0.1
ARM6	1992	v3	3	0.65	30	4 KB unified	0.6
ARM7	1994	v4T	3	0.9	100	0–8 KB unified	1
ARM9E	1999	v5TE	5	1.1	300	0–16 KB I + D	3
ARM11	2002	v6	8	1.25	700	4–64 KB I + D	30
Cortex-A9	2009	v7	8	2.5	1000	16–64 KB I + D	100
Cortex-A7	2011	v7	8	1.9	1500	8–64 KB I + D	40
Cortex-A15	2011	v7	15	3.5	2000	32 KB I + D	240
Cortex-M0 +	2012	v7M	2	0.93	60–250	None	0.3
Cortex-A53	2012	v8	8	2.3	1500	8–64 KB I + D	50
Cortex-A57	2012	v8	15	4.1	2000	48 KB I + 32 KB D	300

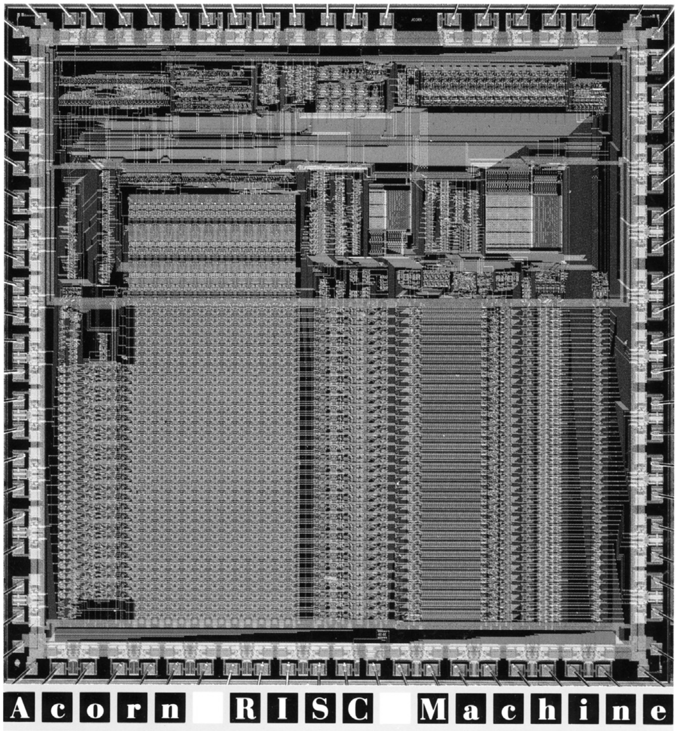


Figure 7.68 **ARM1 die photograph**

(Reproduced with permission from ARM. © 1985 ARM Ltd.)

frequency over three decades and eight revisions of the architecture. Frequency, area, and power will vary with manufacturing process and the goals, schedule, and capabilities of the design team. The representative frequencies are quoted for a fabrication process at the time of product introduction, so much of the frequency gain comes from transistors rather than microarchitecture. The relative size is normalized by the transistor feature size and can vary widely depending on cache size and other factors.

Figure 7.68 shows a die photograph of the ARM1 processor, which contained 25,000 transistors in a three-stage pipeline. If you count carefully, you can observe the 32 bits of the datapath at the bottom. The register file is on the left and the ALU is on the right. At the very left is the program counter; observe that the two least significant bits at the bottom are empty (tied to 0) and the six at the top are different because they are used for status bits. The controller sits on top of the datapath. Some of the rectangular blocks are PLAs implementing control logic. The rectangles around the edge are I/O pads, with tiny gold bond wires visible leading out of the picture.

In 1990, Acorn spun off the processor design team to establish a new company, Advanced RISC Machines (later named ARM Holdings), which began licensing the ARMv3 architecture. The ARMv3 architecture moved the status bits from the PC to the Current Program Status Register and extended the PC to 32 bits. Apple bought a major stake in ARM and used the ARM 610 in the Newton computer, the world's first Personal Digital Assistant (PDA) and one of the first commercial applications of handwriting recognition. Newton proved to be ahead of its time, but it laid the foundation for more successful PDAs and later for smart phones and tablets.

ARM achieved huge success with the ARM7 line in 1994, especially the ARM7TDMI, which became one of the mostly widely used RISC processors in embedded systems over the next 15 years. The ARM7TDMI used the ARMv4T instruction set, which introduced the Thumb instruction set for better code density and defined halfword and signed byte load and store instructions. TDMI stood for **T**humb, JTAG **D**ebug, fast **M**ultiply, and **I**n-Circuit Debug. The various debug features help programmers write code on the hardware and test it from a PC using a simple cable, an important advance at the time. ARM7 used a simple three-stage pipeline with Fetch, Decode, and Execute stages. The processor had a unified cache containing both instructions and data. Because the cache in a pipelined processor is usually busy every cycle fetching instructions, ARM7 stalled memory instructions in the Execute stage to make time for the cache to access the data. Figure 7.69 shows a block diagram of the processor. Rather than manufacturing a chip directly, ARM licensed the processor to other companies that put them into their larger system-on-chip (SoC). Customers could buy the processor as a hard macro (a complete and efficient but inflexible layout that could be dropped directly into a chip) or as a soft macro (Verilog code that could be synthesized by the customer). The ARM7 was used in a vast number of products, including mobile phones, the Apple iPod, Lego Mindstorms NXT, Nintendo game machines, and automobiles. Since then, nearly all mobile phones have been built around ARM processors.

The ARM9E line improved on ARM7 with a five-stage pipeline similar to the one described in this chapter, separate instruction and data caches, and new Thumb and digital signal processing instructions in the ARMv5TE architecture. Figure 7.70 shows a block diagram of the ARM9 containing many of the same components as we encountered in this chapter but adding the multiplier and shifter. The IA/ID/DA/DD signals are the Instruction and Data Address and Data busses to the memory system, and the IAreg is the PC. The next-generation ARM11 extended the pipeline further to eight stages to boost frequency and defined Thumb2 and SIMD instructions.

The ARMv7 instruction set added Advanced SIMD instructions operating on double- and quad-word registers. It also defined a v7-M variant

Sophie Wilson and Steve Furber together designed the ARM1.

Sophie Wilson (1957–) was born in Yorkshire, England, and studied Computer Science at the University of Cambridge. She designed the operating system and wrote the BBC Basic Interpreter for Acorn Computer, and then codesigned the ARM1 and subsequent processors through the ARM7. By 1999, she designed the Firepath SIMD digital signal processor and spun it off as a new company, which Broadcom acquired in 2001. She is presently a Senior Director at Broadcom Corporation and a Fellow of the Royal Society, the Royal Academy of Engineering, the British Computer Society, and the Women's Engineering Society.

(Photograph © Sophie Wilson. Reproduced with permission.)

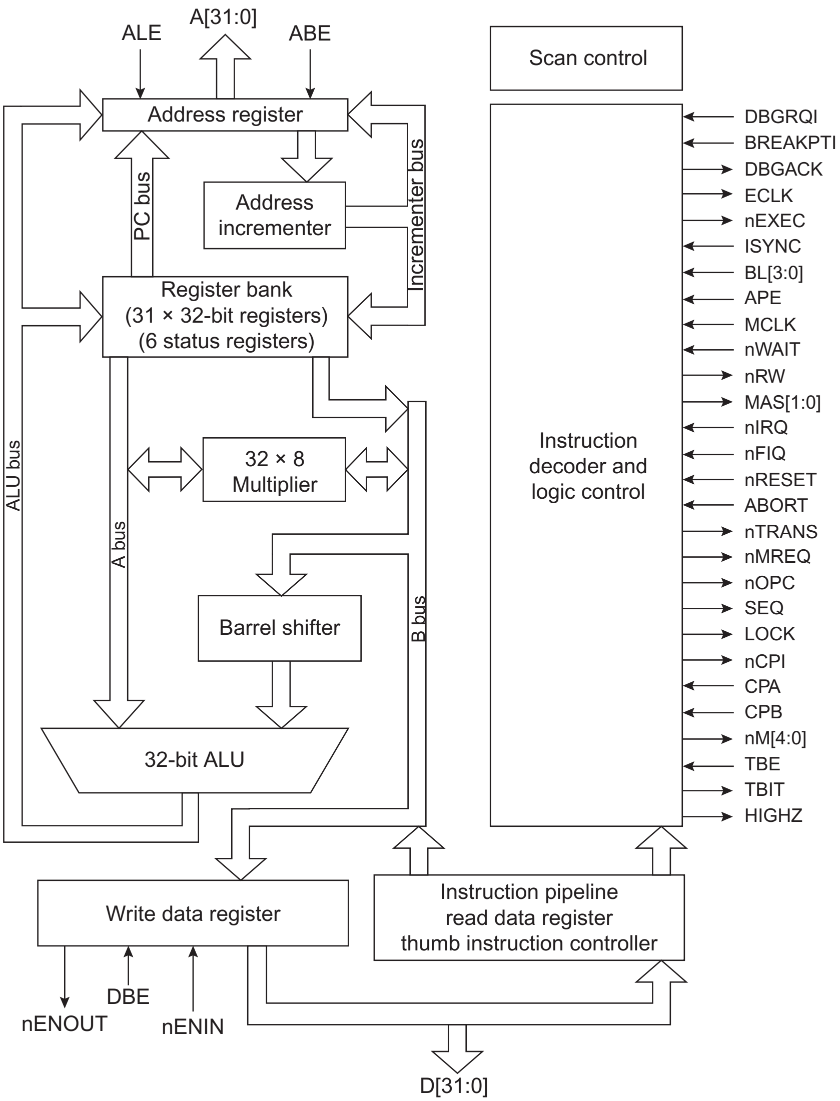

ALE
A[31:0]
ABE
Scan control
Address register
Address incrementer
PC bus
Incrementer bus
Register bank
(31 × 32-bit registers)
(6 status registers)
ALU bus
A bus
B bus
32 × 8 Multiplier
Barrel shifter
32-bit ALU
Instruction decoder and logic control
DBGRQI
BREAKPTI
DBGACK
ECLK
nEXEC
ISYNC
BL[3:0]
APE
MCLK
nWAIT
nRW
MAS[1:0]
nIRQ
nFIQ
nRESET
ABORT
nTRANS
nMREQ
nOPC
SEQ
LOCK
nCPI
CPA
CPB
nM[4:0]
TBE
TBIT
HIGHZ
Write data register
Instruction pipeline
read data register
thumb instruction controller
nENOUT
DBE
nENIN
D[31:0]

Figure 7.69 **ARM7 block diagram** (Reproduced with permission from ARM. © 1998 ARM Ltd.)

Steve Furber (1953–) was born in Manchester, England, and received a PhD in aerodynamics from the University of Cambridge. He joined Acorn Computer, where he codesigned the BBC Micro and ARM1 microprocessor for Acorn Computer. In 1990, he joined the faculty of the University of Manchester, where his research has focused on asynchronous computing and neural systems.

(Photograph © 2012 The University of Manchester. Reproduced with permission.)

supporting only Thumb instructions. ARM introduced the Cortex-A and Cortex-M families of processors. The Cortex-A family of high-performance processors are now used in virtually all smart phones and tablets. The Cortex-M family, running the Thumb instruction set, are tiny and inexpensive microcontrollers used in embedded systems. For example, the Cortex-M0+ uses a two-stage pipeline and only 12,000 gates, compared with hundreds of thousands in an A-series processor. It costs well under a dollar as a stand-alone chip, or under a penny when integrated

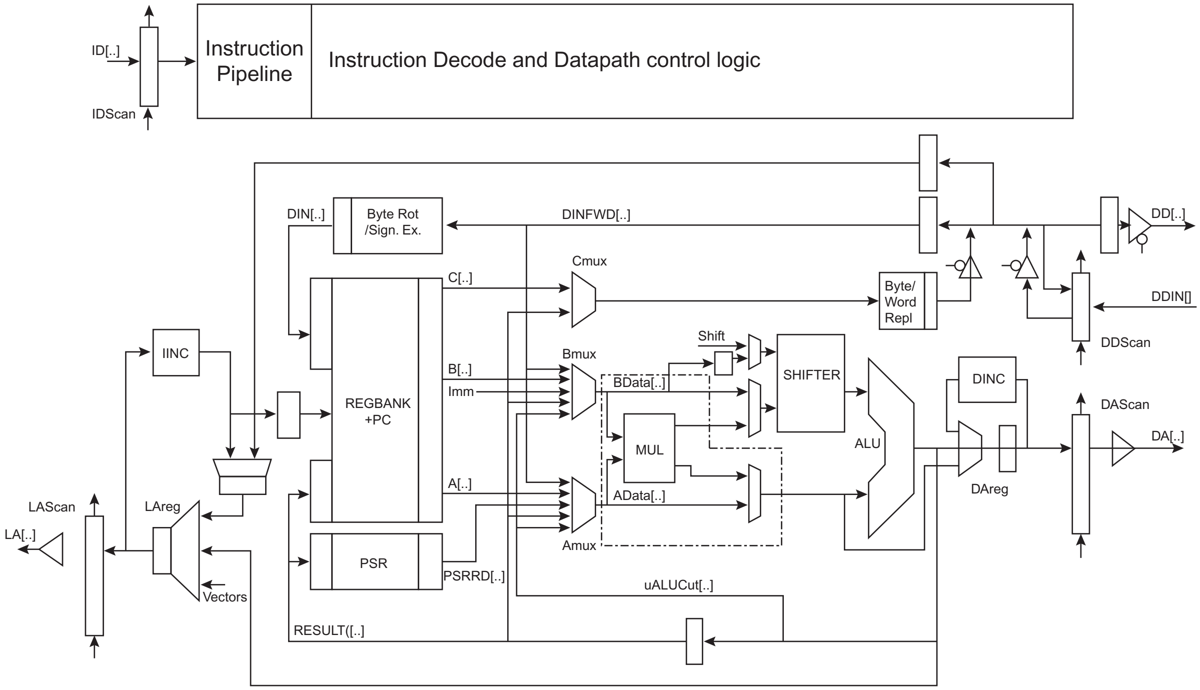

ID[..]
IDScan
Instruction Pipeline
Instruction Decode and Datapath control logic
DIN[..]
Byte Rot /Sign. Ex.
DINFWD[..]
DD[..]
Cmux
C[..]
Byte/ Word Repl
DDIN[]
DDScan
Shift
IINC
Bmux
B[..]
Imm
BData[..]
SHIFTER
DINC
DAScan
REGBANK +PC
MUL
ALU
DA[..]
A[..]
DAreg
AData[..]
LAScan
LAreg
LA[..]
Amux
PSR
PSRRD[..]
uALUCut[..]
Vectors
RESULT([..]

Figure 7.70 ARM9 block diagram

(Reproduced with permission from the ARM9TDMI Technical Reference Manual. © 1999 ARM Ltd.)

on a larger SoC. The power consumption is roughly 3 μW/MHz, so the processor powered by a watch battery could run continuously for nearly a year at 10 MHz.

Higher-end ARMv7 processors captured the cell phone and tablet markets. The Cortex-A9 was widely used in mobile phones, often as part of a dual-core SoC containing two Cortex-A9 processors, a graphics accelerator, a cellular modem, and other peripherals. Figure 7.71 shows a block diagram of the Cortex-A9. The processor decodes two instructions per cycle, performs register renaming, and issues them to out-of-order execution units.

Energy efficiency and performance are both critical for mobile devices, so ARM has been promoting the big.LITTLE architecture combining several high-performance "big" cores for peak workloads with energy-efficient "LITTLE" cores that handle most routine processes. For example, the Samsung Exynos 5 Octa in the Galaxy S5 phone contains four Cortex-A15 big cores running up to 2.1 GHz and four Cortex-A7 LITTLE cores running at up to 1.5 GHz. Figure 7.72 shows pipeline diagrams for the two types of cores. The Cortex-A7 is an in-order processor that can decode and issue up to one memory instruction and

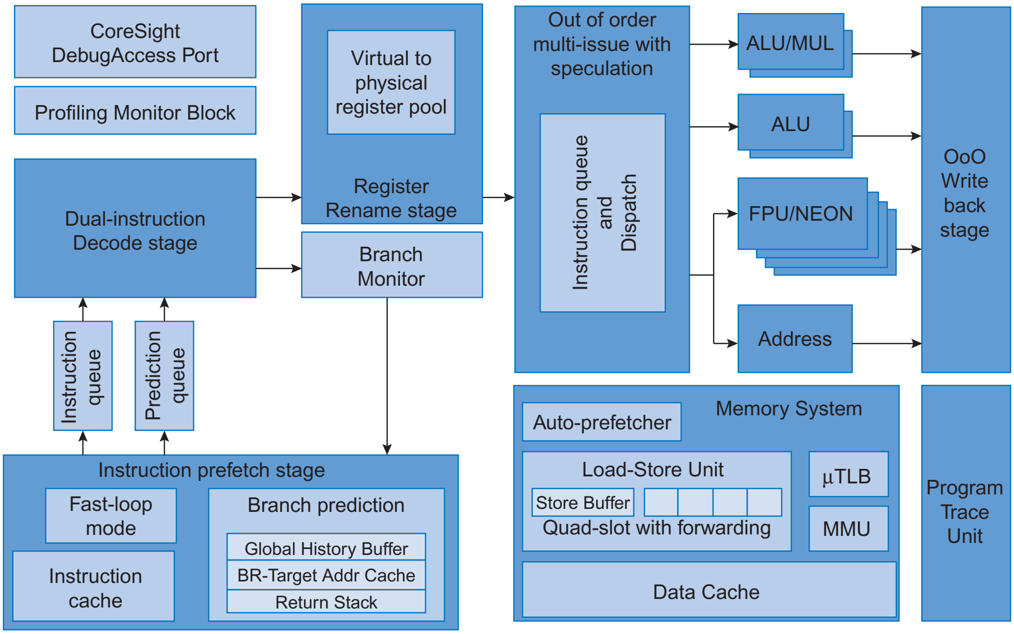

CoreSight DebugAccess Port
Profiling Monitor Block
Dual-instruction Decode stage
Instruction queue
Prediction queue
Virtual to physical register pool
Register Rename stage
Branch Monitor
Out of order multi-issue with speculation
Instruction queue and Dispatch
ALU/MUL
ALU
FPU/NEON
Address
OoO Write back stage
Instruction prefetch stage
Fast-loop mode
Instruction cache
Branch prediction
Global History Buffer
BR-Target Addr Cache
Return Stack
Memory System
Auto-prefetcher
Load-Store Unit
Store Buffer
Quad-slot with forwarding
µTLB
MMU
Data Cache
Program Trace Unit

Figure 7.71 **Cortex-A9 block diagram**

(This image has been sourced by the authors and does not imply ARM endorsement.)

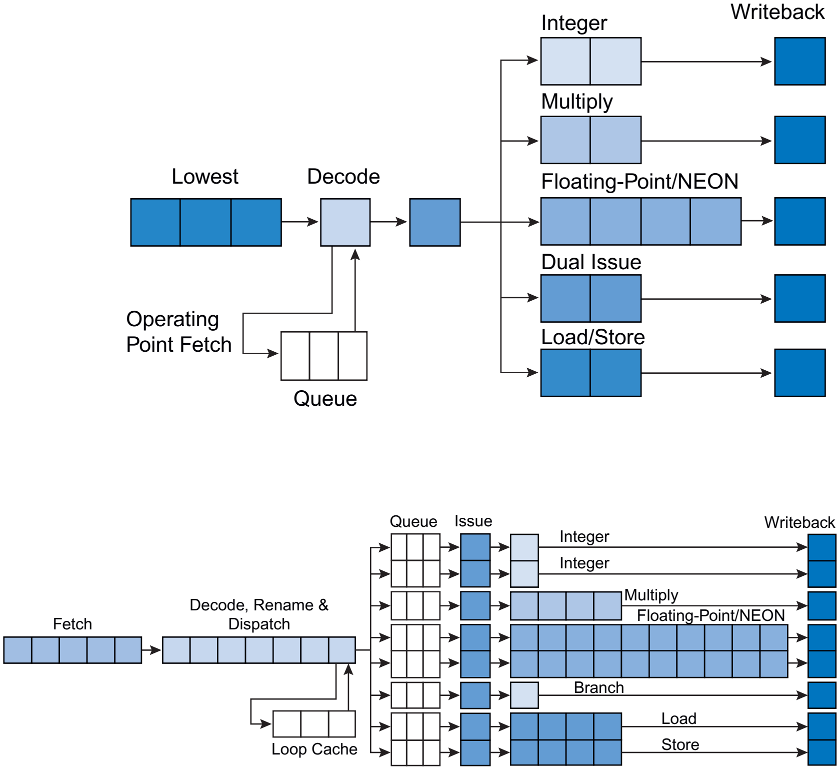

Lowest
Decode
Operating Point Fetch
Queue
Integer
Multiply
Floating-Point/NEON
Dual Issue
Load/Store
Writeback
Fetch
Decode, Rename & Dispatch
Loop Cache
Queue
Issue
Integer
Integer
Multiply
Floating-Point/NEON
Branch
Load
Store
Writeback

Figure 7.72 **Cortex-A7 and -A15 block diagrams**

(This image has been sourced by the authors and does not imply ARM endorsement.)

one other instruction each cycle. The Cortex-A15 is a much more complex out-of-order processor that can decode up to three instructions each cycle. The pipeline length almost doubles to handle the complexity and boost clock speed, so a more accurate branch predictor is necessary to compensate for the larger branch misprediction penalty. The Cortex-A15 delivers approximately 2.5x the performance of the Cortex-A7, but at 6x the power. Smart phones can only run the big cores briefly before the chip will begin to overheat and throttle itself back.

The ARMv8 architecture is a streamlined 64-bit architecture. ARM's Cortex-A53 and -A57 have pipelines similar to the Cortex-A7 and -A15, respectively, but boost the registers and datapaths to 64 bits to handle ARMv8. Apple popularized the 64-bit architecture in 2013, when it introduced its own implementation in the iPhone and iPad.

7.9 SUMMARY

This chapter has described three ways to build processors, each with different performance and cost trade-offs. We find this topic almost magical: how can such a seemingly complicated device as a microprocessor actually be simple enough to fit in a half-page schematic? Moreover, the inner workings, so mysterious to the uninitiated, are actually reasonably straightforward.

The microarchitectures have drawn together almost every topic covered in the text so far. Piecing together the microarchitecture puzzle illustrates the principles introduced in previous chapters, including the design of combinational and sequential circuits (covered in Chapters 2 and 3), the application of many of the building blocks (described in Chapter 5), and the implementation of the ARM architecture (introduced in Chapter 6). The microarchitectures can be described in a few pages of HDL using the techniques from Chapter 4.

Building the microarchitectures has also heavily used our techniques for managing complexity. The microarchitectural abstraction forms the link between the logic and architecture abstractions, forming the crux of this book on digital design and computer architecture. We also use the abstractions of block diagrams and HDL to succinctly describe the arrangement of components. The microarchitectures exploit regularity and modularity, reusing a library of common building blocks such as ALUs, memories, multiplexers, and registers. Hierarchy is used in numerous ways. The microarchitectures are partitioned into the datapath and control units. Each of these units is built from logic blocks, which can be built from gates, which in turn can be built from transistors using the techniques developed in the first five chapters.

This chapter has compared single-cycle, multicycle, and pipelined microarchitectures for the ARM processor. All three microarchitectures implement the same subset of the ARM instruction set and have the same architectural state. The single-cycle processor is the most straightforward and has a CPI of 1.

The multicycle processor uses a variable number of shorter steps to execute instructions. It thus can reuse the ALU, rather than requiring several adders. However, it does require several nonarchitectural registers to store results between steps. The multicycle design in principle could be faster, because not all instructions must be equally long. In practice, it is generally slower, because it is limited by the slowest steps and by the sequencing overhead in each step.

The pipelined processor divides the single-cycle processor into five relatively fast pipeline stages. It adds pipeline registers between the stages to separate the five instructions that are simultaneously executing. It nominally has a CPI of 1, but hazards force stalls or flushes that increase the CPI slightly. Hazard resolution also costs some extra hardware and design complexity. The clock period ideally could be five times shorter than that of the single-cycle processor. In practice, it is not that short, because it is limited by the slowest stage and by the sequencing overhead in each stage. Nevertheless, pipelining provides substantial performance benefits. All modern high-performance microprocessors use pipelining today.

Although the microarchitectures in this chapter implement only a subset of the ARM architecture, we have seen that supporting more instructions involves straightforward enhancements of the datapath and controller.

A major limitation of this chapter is that we have assumed an ideal memory system that is fast and large enough to store the entire program and data. In reality, large fast memories are prohibitively expensive. The next chapter shows how to get most of the benefits of a large fast memory with a small fast memory that holds the most commonly used information and one or more larger but slower memories holding the rest of the information.

Exercises

Exercise 7.1 Suppose that one of the following control signals in the single-cycle ARM processor has a *stuck-at-0 fault*, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *RegW*

(b) *ALUOp*

(c) *MemW*

Exercise 7.2 Repeat Exercise 7.1, assuming that the signal has a stuck-at-1 fault.

Exercise 7.3 Modify the single-cycle ARM processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.13 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Tables 7.2 and 7.3 to show the changes to the Main Decoder and ALU Decoder. Describe any other changes that are required.

(a) `TST`

(b) `LSL`

(c) `CMN`

(d) `ADC`

Exercise 7.4 Repeat Exercise 7.3 for the following ARM instructions.

(a) `EOR`

(b) `LSR`

(c) `TEQ`

(d) `RSB`

Exercise 7.5 ARM includes `LDR` with post-indexing, which updates the base register after completing the load. `LDR Rd,[Rn],Rm` is equivalent to the following two instructions:

```
LDR  Rd,[Rn]
ADD  Rn,Rn,Rm
```

Repeat Exercise 7.3 for `LDR` with post-indexing. Is it possible to add the instruction without modifying the register file?

Exercise 7.6 ARM includes `LDR` with pre-indexing, which updates the base register after completing the load. `LDR Rd,[Rn,Rm]!` is equivalent to the following two instructions:

```
LDR  Rd,[Rn, Rm]
ADD  Rn,Rn,Rm
```

Repeat Exercise 7.3 for `LDR` with pre-indexing. Is it possible to add the instruction without modifying the register file?

Exercise 7.7 Your friend is a crack circuit designer. She has offered to redesign one of the units in the single-cycle ARM processor to have half the delay. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be?

Exercise 7.8 Consider the delays given in Table 7.5. Ben Bitdiddle builds a prefix adder that reduces the ALU delay by 20 ps. If the other element delays stay the same, find the new cycle time of the single-cycle ARM processor and determine how long it takes to execute a benchmark with 100 billion instructions.

Exercise 7.9 Modify the HDL code for the single-cycle ARM processor, given in Section 7.6.1, to handle one of the new instructions from Exercise 7.3. Enhance the testbench, given in Section 7.6.3, to test the new instruction.

Exercise 7.10 Repeat Exercise 7.9 for the new instructions from Exercise 7.4.

Exercise 7.11 Suppose one of the following control signals in the multicycle ARM processor has a stuck-at-0 fault, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) $RegSrc_1$

(b) $AdrSrc$

(c) $NextPC$

Exercise 7.12 Repeat Exercise 7.11, assuming that the signal has a stuck-at-1 fault.

Exercise 7.13 Modify the multicycle ARM processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.30 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Figure 7.41 to show the changes to the controller FSM. Describe any other changes that are required.

(a) `ASR`

(b) `TST`

(c) `SBC`

(d) `ROR`

Exercise 7.14 Repeat Exercise 7.13 for the following ARM instructions.

(a) `BL`

(b) `LDR` (with positive or negative immediate offset)

(c) `LDRB` (with positive immediate offset only)

(d) `BIC`

Exercise 7.15 Repeat Exercise 7.5 for the multicycle ARM processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.16 Repeat Exercise 7.6 for the multicycle ARM processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.17 Repeat Excercise 7.7 for the multicycle ARM processor. Assume the instruction mix of Example 7.5.

Exercise 7.18 Repeat Exercise 7.8 for the multicycle ARM processor. Assume the instruction mix of Example 7.5.

Exercise 7.19 Your friend, the crack circuit designer, has offered to redesign one of the units in the multicycle ARM processor to be much faster. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.20 Goliath Corp claims to have a patent on a three-ported register file. Rather than fighting Goliath in court, Ben Bitdiddle designs a new register file that has only a single read/write port (like the combined instruction and data memory). Redesign the ARM multicycle datapath and controller to use his new register file.

Exercise 7.21 Suppose the multicycle ARM processor has the component delays given in Table 7.5. Alyssa P. Hacker designs a new register file that has 40% less power but twice as much delay. Should she switch to the slower but lower power register file for her multicycle processor design?

Exercise 7.22 What is the CPI of the redesigned multicycle ARM processor from Exercise 7.20? Use the instruction mix from Example 7.5.

Exercise 7.23 How many cycles are required to run the following program on the multicycle ARM processor? What is the CPI of this program?

```
    MOV  R0,  #5            ; result = 5
    MOV  R1,  #0            ; R1 = 0
L1
    CMP  R0,  R1
    BEQ  DONE               ; if result > 0, loop
    SUB   R0,  R0,  #1      ; result = result-1
    B     L1
DONE
```

Exercise 7.24 Repeat Exercise 7.23 for the following program.

```
    MOV  R0,  #0            ; i = 0
    MOV  R1,  #0            ; sum = 0
    MOV  R2,  #10           ; R2 = 10

LOOP
    CMP  R2,  R0            ; R2 == R0?
    BEQ  L2
    ADD  R1,  R1,  R0       ; sum = sum + i
    ADD  R0,  R0,  #1       ; increment i
    B
    LOOP
L2
```

Exercise 7.25 Write HDL code for the multicycle ARM processor. The processor should be compatible with the following top-level module. The `mem` module is used to hold both instructions and data. Test your processor using the testbench from Section 7.6.3.

```
module top(input   logic          clk, reset,
           output  logic  [31:0]  WriteData, Adr,
           output  logic          MemWrite);

  logic [31:0] ReadData;

  // instantiate processor and shared memory
  arm arm(clk, reset, MemWrite, Adr,
          WriteData, ReadData);
  mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule

module mem(input   logic          clk, we,
           input   logic  [31:0]  a, wd,
           output  logic  [31:0]  rd);

  logic [31:0] RAM[63:0];
  initial
     $readmemh("memfile.dat",RAM);
```

```
    assign rd = RAM[a[31:2]];  // word aligned

    always_ff @(posedge clk)
      if (we) RAM[a[31:2]] <= wd;
  endmodule
```

Exercise 7.26 Extend your HDL code for the multicycle ARM processor from Exercise 7.25 to handle one of the new instructions from Exercise 7.14. Enhance the testbench to test the new instruction.

Exercise 7.27 Repeat Exercise 7.26 for one of the new instructions from Exercise 7.13.

Exercise 7.28 The pipelined ARM processor is running the following code snippet. Which registers are being written, and which are being read on the fifth cycle? Recall that the pipelined ARM processor has a Hazard Unit.

```
MOV  R1,  #42
SUB  R0,  R1,  #5
LDR  R3,  [R0, #18]
STR  R4,  [R1, #63]
ORR  R2,  R0,  R3
```

Exercise 7.29 Repeat Exercise 7.28 for the following ARM code snippet.

```
ADD  R0,  R4,  R5
SUB  R1,  R6,  R7
AND  R2,  R0,  R1
ORR  R3,  R2,  R5
LSL  R4,  R2,  R3
```

Exercise 7.30 Using a diagram similar to Figure 7.53, show the forwarding and stalls needed to execute the following instructions on the pipelined ARM processor.

```
ADD  R0,  R4,  R9
SUB  R0,  R0,  R2
LDR  R1,  [R0, #60]
AND  R2,  R1, R0
```

Exercise 7.31 Repeat Exercise 7.30 for the following instructions.

```
ADD  R0,  R11,  R5
LDR  R2,  [R1, #45]
SUB  R5,  R0,  R2
AND  R5,  R2,  R5
```

Exercise 7.32 How many cycles are required for the pipelined ARM processor to issue all of the instructions for the program in Exercise 7.24? What is the CPI of the processor on this program?

Exercise 7.33 Repeat Exercise 7.32 for the instructions of the program in Exercise 7.23.

Exercise 7.34 Explain how to extend the pipelined ARM processor to handle the `EOR` instruction.

Exercise 7.35 Explain how to extend the pipelined processor to handle the `CMN` instruction.

Exercise 7.36 Section 7.5.3 points out that the pipelined processor performance might be better if branches take place during the Decode stage rather than the Execute stage. Show how to modify the pipelined processor from Figure 7.58 to branch in the Decode stage. How do the stall, flush, and forwarding signals change? Redo Examples 7.7 and 7.8 to find the new CPI, cycle time, and overall time to execute the program.

Exercise 7.37 Your friend, the crack circuit designer, has offered to redesign one of the units in the pipelined ARM processor to be much faster. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.38 Consider the delays from Table 7.5. Now suppose that the ALU were 20% faster. Would the cycle time of the pipelined ARM processor change? What if the ALU were 20% slower?

Exercise 7.39 Suppose the ARM pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

Exercise 7.40 Write HDL code for the pipelined ARM processor. The processor should be compatible with the top-level module from HDL Example 7.13. It should support the seven instructions described in this chapter: `ADD`, `SUB`, `AND`, `ORR` (with register and immediate addressing modes but no shifts), `LDR`, `STR` (with positive immediate offset), and `B`. Test your design using the testbench from HDL Example 7.12.

Exercise 7.41 Design the Hazard Unit shown in Figure 7.58 for the pipelined ARM processor. Use an HDL to implement your design. Sketch the hardware that a synthesis tool might generate from your HDL.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 7.1 Explain the advantages of pipelined microprocessors?

Question 7.2 If additional pipeline stages allow a processor to go faster, why don't processors have 100 pipeline stages?

Question 7.3 Describe what a hazard is in a microprocessor and explain ways in which it can be resolved. What are the pros and cons of each way?

Question 7.4 Describe the concept of a superscalar processor and its pros and cons?

This page intentionally left blank

8

Memory Systems

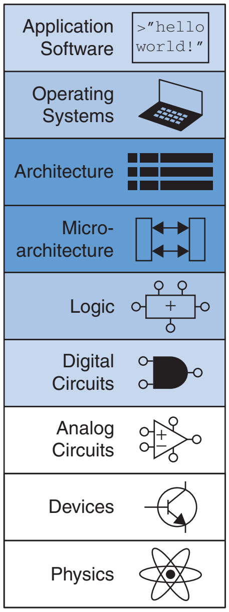
8.1 Introduction
8.2 Memory System Performance Analysis
8.3 Caches
8.4 Virtual Memory
8.5 Summary
Epilogue
Exercises
Interview Questions

8.1 INTRODUCTION

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. However, this would be true only for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. DRAM memories are currently 10 to 100 times slower than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. This chapter investigates memory systems and considers tradeoffs of speed, capacity, and cost.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle ARM processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

The major issues in memory system design can be broadly explained using a metaphor of books in a library. A library contains many books on the shelves. If you were writing a term paper on the meaning of dreams, you might go to the library[1] and pull Freud's *The Interpretation of Dreams* off the shelf and bring it to your cubicle. After skimming it, you might put it back and pull out Jung's *The Psychology of the Unconscious*. You might then go back for another quote from *Interpretation of Dreams*, followed by yet another trip to the stacks for Freud's *The Ego and the Id*. Pretty soon



---

[1] We realize that library usage is plummeting among college students because of the Internet. But we also believe that libraries contain vast troves of hard-won human knowledge that are not electronically available. We hope that Web searching does not completely displace the art of library research.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00008-2
© 2013 Elsevier, Inc. All rights reserved.

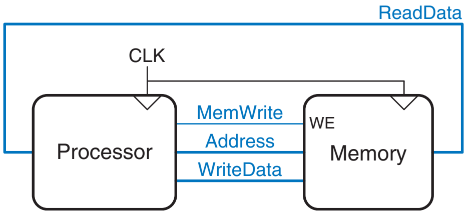


Figure 8.1 **The memory interface**

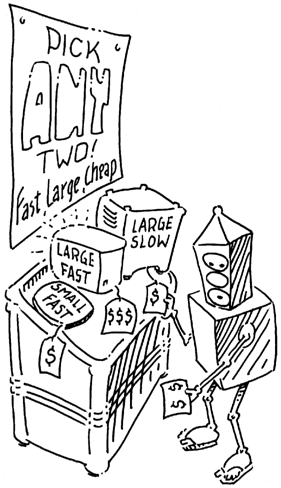
you would get tired of walking from your cubicle to the stacks. If you are clever, you would save time by keeping the books in your cubicle rather than schlepping them back and forth. Furthermore, when you pull a book by Freud, you could also pull several of his other books from the same shelf.

This metaphor emphasizes the principle, introduced in Section 6.2.1, of making the common case fast. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

The library itself makes the common case fast by using these principles of locality. The library has neither the shelf space nor the budget to accommodate all of the books in the world. Instead, it keeps some of the lesser-used books in deep storage in the basement. Also, it may have an interlibrary loan agreement with nearby libraries so that it can offer more books than it physically carries.

In summary, you obtain the benefits of both a large collection and quick access to the most commonly used books through a hierarchy of storage. The most commonly used books are in your cubicle. A larger collection is on the shelves. And an even larger collection is available, with advanced notice, from the basement and other libraries. Similarly, memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data.

Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (DRAM) and static RAM (SRAM). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory only has two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by combining a fast small cheap memory and a slow large cheap memory. The fast memory stores the most commonly used data and instructions, so on average the memory system appears fast. The large memory stores the remainder of the data and instructions, so the overall capacity is large. The combination of two cheap memories is much less



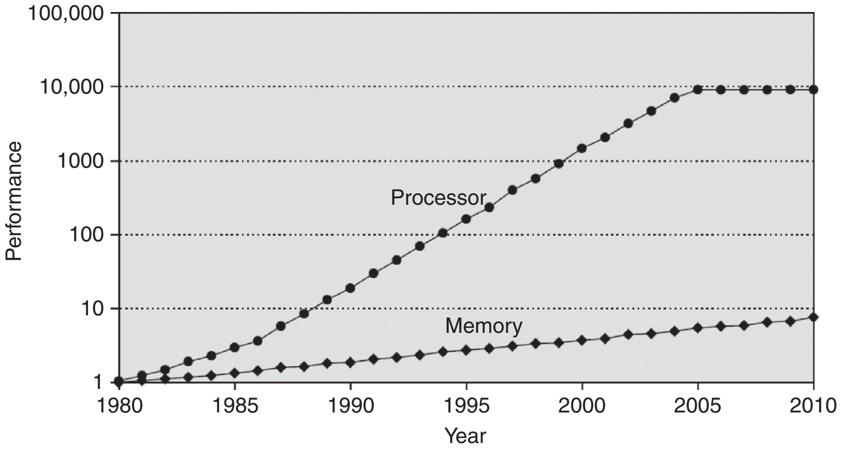


Figure 8.2 Diverging processor and memory performance Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011.

expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

Computer memory is generally built from DRAM chips. In 2015, a typical PC had a *main memory* consisting of 8 to 16 GB of DRAM, and DRAM cost about $7 per gigabyte (GB). DRAM prices have declined at about 25% per year for the last three decades, and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 25 to 50% per year, as shown in Figure 8.2. The plot shows memory (DRAM) and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. But performance has diverged since then, with memories badly lagging.[2]

DRAM could keep up with processors in the 1970s and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed, because SRAM is inherently faster than DRAM, and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip. In 2015, on-chip SRAM costs were on the order of $5,000/GB, but the cache is relatively small (kilobytes to several megabytes), so the overall cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data."

---

[2] Although recent single processor performance has remained approximately constant, as shown in Figure 8.2 for the years 2005–2010, the increase in multi-core systems (not depicted on the graph) only worsens the gap between processor and memory performance.

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that does not fit in main memory. In 2015, a hard disk drive (HDD), built using magnetic storage, cost less than \$0.05/GB and had an access time of about 5 ms. Hard disk costs have decreased at 60%/year but access times scarcely improved. Solid state drives (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs. SSDs have been used by niche markets for over two decades, and they were introduced into the mainstream market in 2007. SSDs overcome some of the mechanical failures of HDDs, but they cost about ten times as much at \$0.40/GB.

The hard drive provides an illusion of more capacity than actually exists in the main memory. It is thus called virtual memory. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called physical memory, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs, access times, and bandwidth in 2015 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems.

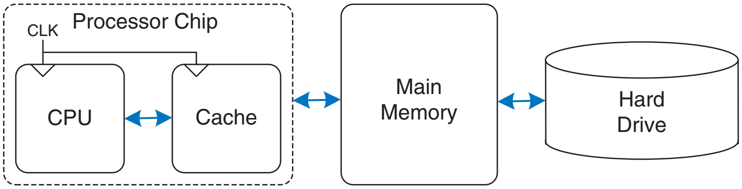
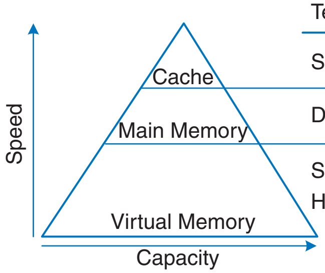


Figure 8.3 **A typical memory hierarchy**



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	\$5,000	0.5	25+
DRAM	\$7	10–50	10
SSD	\$0.40	20,000	0.5
HDD	\$0.05	5,000,000	0.75

Figure 8.4 **Memory hierarchy components, with typical characteristics in 2015**

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time*. Miss and hit rates are calculated as:

$$
\begin{aligned}
Miss\ Rate &= \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate \\
Hit\ Rate &= \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate
\end{aligned}
\tag{8.1}
$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution: The miss rate is $750/2000 = 0.375 = 37.5\%$. The hit rate is $1250/2000 = 0.625 = 1 - 0.375 = 62.5\%$.

Average memory access time (*AMAT*) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \tag{8.2}$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates in Table 8.1?

Solution: The average memory access time is $1 + 0.1(100) = 11$ cycles.

Table 8.1 **Access times and miss rates**

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution: If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

Gene Amdahl, 1922–. Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation, and later went on to found three companies, including one called Amdahl Corporation in 1970.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's performance is due to loads and stores, a tenfold memory system improvement only means a 1.82-fold improvement in program performance. This general principle is called *Amdahl's Law*, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is data found? and (3) What data is replaced to make room for new data when the cache is full?

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

Cache: a hiding place especially for concealing and preserving provisions or implements.

– Merriam Webster Online Dictionary, 2015. www.merriam-webster.com

8.3.1 What Data is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block* or *cache line*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the same variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How is Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider an ARM memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . ., 0xFFFFFFE4 all

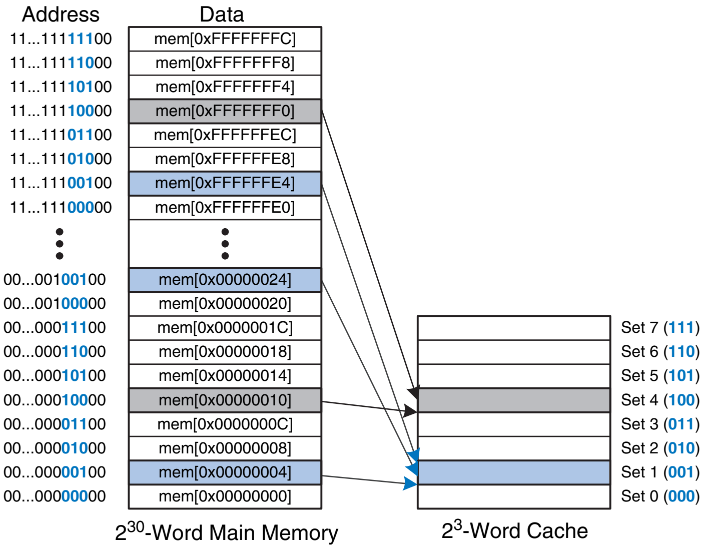

Address
Data
11...11111100
11...11111000
11...11110100
11...11110000
11...11101100
11...11101000
11...11100100
11...11100000
00...00100100
00...00100000
00...00011100
00...00011000
00...00010100
00...00010000
00...00001100
00...00001000
00...00000100
00...00000000
mem[0xFFFFFFFC]
mem[0xFFFFFFF8]
mem[0xFFFFFFF4]
mem[0xFFFFFFF0]
mem[0xFFFFFFEC]
mem[0xFFFFFFE8]
mem[0xFFFFFFE4]
mem[0xFFFFFFE0]
mem[0x00000024]
mem[0x00000020]
mem[0x0000001C]
mem[0x00000018]
mem[0x00000014]
mem[0x00000010]
mem[0x0000000C]
mem[0x00000008]
mem[0x00000004]
mem[0x00000000]
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)
2^30-Word Main Memory
2^3-Word Cache

Figure 8.5 **Mapping of main memory to a direct mapped cache**

map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, . . . , 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution: The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, . . . , 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the *byte offset*, because they indicate the byte within the word. The next three bits are called the *set bits*, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

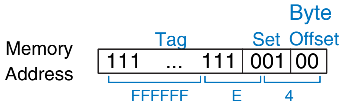

Memory
Address
Tag
Set
Byte
Offset
111 ... 111
001
00
FFFFFF
E
4

Figure 8.6 **Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5**

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution: A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks the tag and valid bits. If the tag matches the most significant 27 bits of the

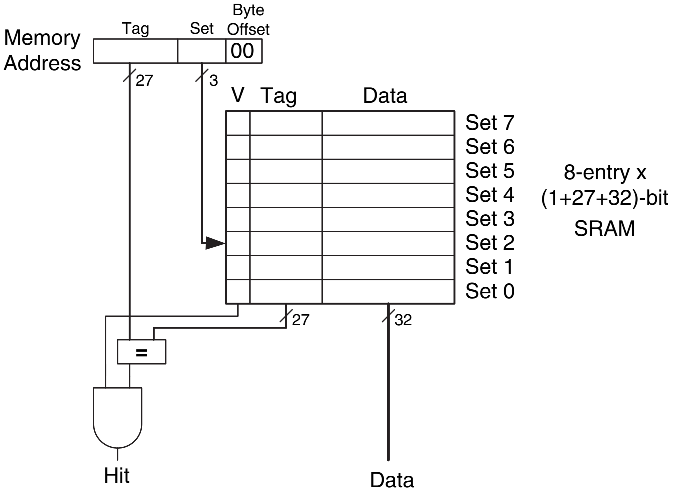

Memory
Address
Tag
Set
Byte
Offset
00
27
3
V
Tag
Data
Set 7
Set 6
Set 5
Set 4
Set 3
Set 2
Set 1
Set 0
8-entry x
(1+27+32)-bit
SRAM
27
32
=
Hit
Data

Figure 8.7 **Direct mapped cache with 8 sets**

address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in ARM assembly code. Assume that the cache is initially empty. What is the miss rate?

```
       MOV R0, #5
       MOV R1, #0
LOOP   CMP R0, #0
       BEQ DONE
       LDR R2, [R1, #4]
       LDR R3, [R1, #12]
       LDR R4, [R1, #8]
       SUB R0, R0, #1
       B   LOOP
DONE
```

Solution: The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.

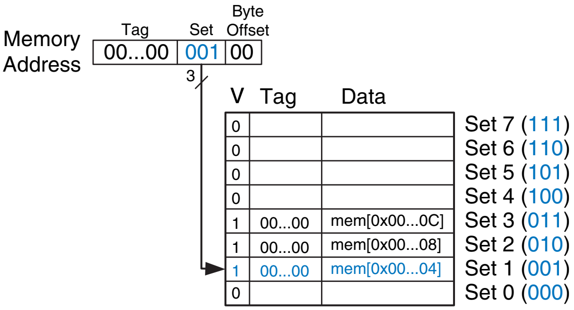

Memory Address
Tag
Set
Byte Offset
00...00
001
00
3
V
Tag
Data
0
0
0
0
1
00...00
mem[0x00...0C]
1
00...00
mem[0x00...08]
1
00...00
mem[0x00...04]
0
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)

Figure 8.8 **Direct mapped cache contents**

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict. Example 8.7 illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
       MOV R0, #5
       MOV R1, #0
LOOP   CMP R0, #0
       BEQ DONE
       LDR R2, [R1, #0x4]
       LDR R3, [R1, #0x24]
       SUB R0, R0, #1
       B   LOOP
DONE
```

Solution: Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict, and the miss rate is 100%.

Multi-way Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the *N* blocks

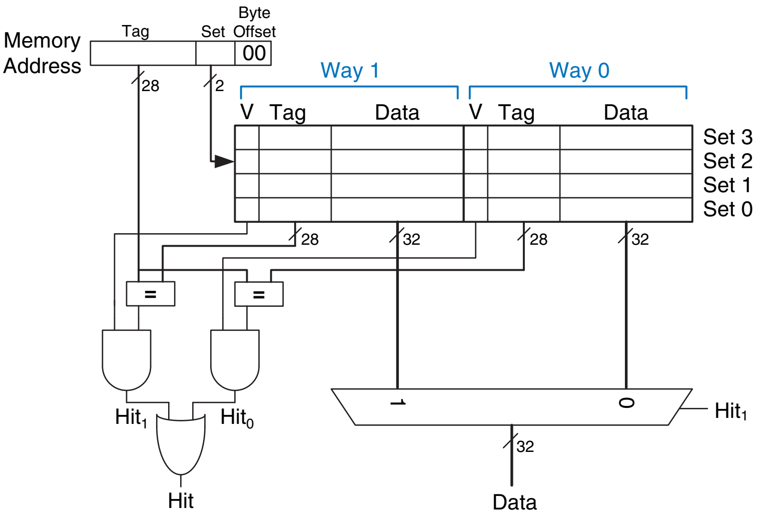

Memory Address
Tag
Set
Byte Offset
00
28
2
Way 1
Way 0
V Tag Data
V Tag Data
Set 3
Set 2
Set 1
Set 0
28
32
28
32
=
=
Hit1
Hit0
Hit
1
0
Hit1
32
Data

Figure 8.9 **Two-way set associative cache**

in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the *degree of associativity* of the cache.

Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution: Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...24]	1	00...10	mem[0x00...04]	Set 1
0			0			Set 0

Figure 8.10 **Two-way set associative cache contents**

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made, because the data could be in any block. Similarly, an 8:1

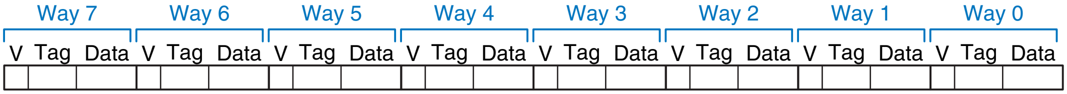


Figure 8.11 Eight-block fully associative cache

multiplexer chooses the proper data if a hit occurs. Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality, because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache

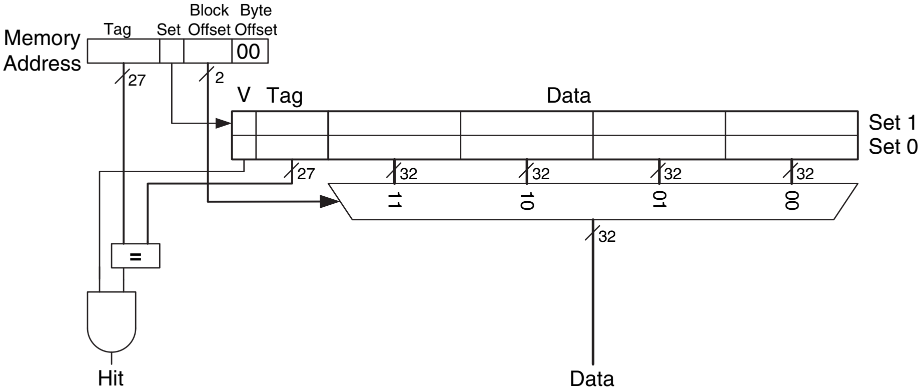


Figure 8.12 Direct mapped cache with two sets and a four-word block size

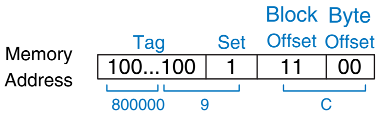

Memory
Address
Tag
Set
Block Offset
Byte Offset
100...100
1
11
00
800000
9
C

Figure 8.13 **Cache fields for address 0x8000009C when mapping to the cache of Figure 8.12**

is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution: Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

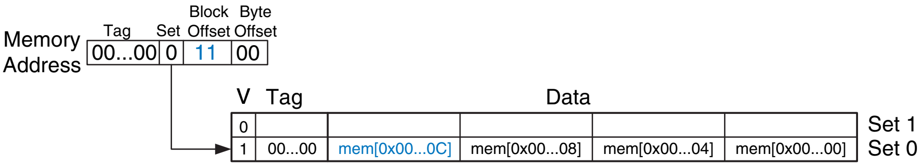

Memory
Address
Tag
Set
Block Offset
Byte Offset
00...00
0
11
00
V
Tag
Data
0
1
00...00
mem[0x00...0C]
mem[0x00...08]
mem[0x00...04]
mem[0x00...00]
Set 1
Set 0

Figure 8.14 **Cache contents with a block size *b* of four words**

Putting it All Together

Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array

Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of 2. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity N usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size b takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and therefore could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (LRU) replacement policy.

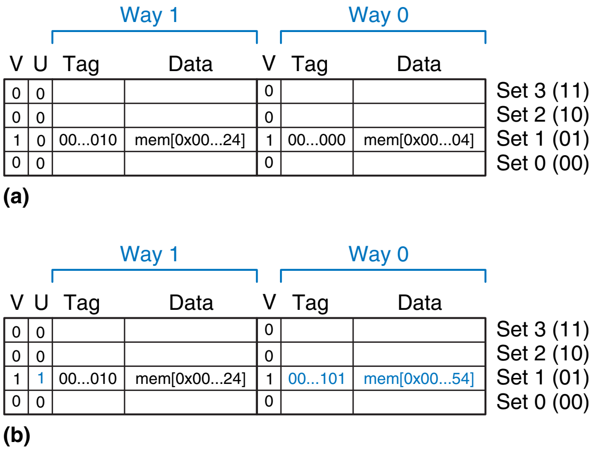
In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used. Upon replacement, the new block replaces a random block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
MOV R0, #0
LDR R1, [R0, #4]
LDR R2, [R0, #0x24]
LDR R3, [R0, #0x54]
```

Solution: The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). *U* = 0 indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b). The use bit *U* is set to 1 to indicate that data in way 1 was the least recently used.



(a)

Way 1				Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

(b)

Way 1				Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)

Figure 8.15 **Two-way associative cache with LRU replacement**

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

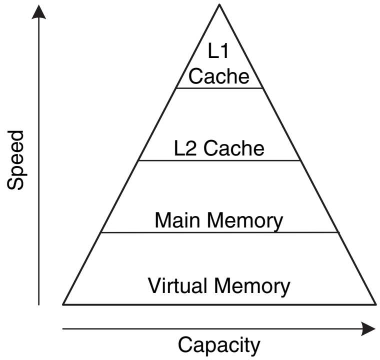


Figure 8.16 **Memory hierarchy with two levels of cache**

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and therefore have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use at least two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger, and therefore slower, than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy, because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution: Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows: 1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles

The L2 miss rate is high because it receives only the “hard” memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

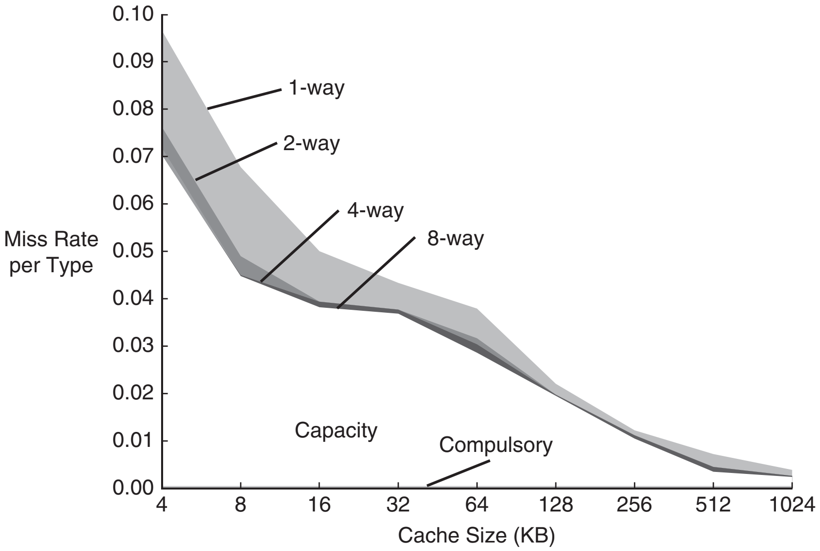


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark (Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.)

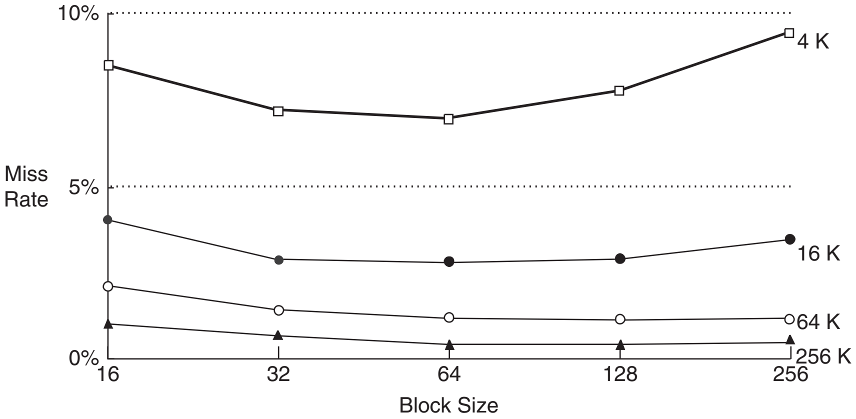


Figure 8.18 **Miss rate versus block size and cache size on SPEC92 benchmark** (Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.)

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4-KB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (*D*) is associated with each cache block. *D* is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back, because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
MOV R5, #0
STR R1, [R5]
STR R2, [R5, #12]
STR R3, [R5, #8]
STR R4, [R5, #4]
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.3.5 The Evolution of ARM Caches*

Table 8.3 traces the evolution of cache organizations used by the ARM processor from 1985 to 2012. The major trends are the introduction of multiple levels of cache, larger cache capacity, and separation of instruction and data L1 caches. These trends are driven by the growing disparity between CPU frequency and main memory speed and the decreasing cost

Table 8.3 **ARM cache evolution**

Year	CPU	MHz	L1 Cache	L2 Cache
1985	ARM1	8	None	None
1992	ARM6	30	4 KB, unified	None
1994	ARM7	100	8 KB, unified	None
1999	ARM9E	300	0–128 KB, I/D	None
2002	ARM11	700	4–64 KB, I/D	0–128 KB, off-chip
2009	Cortex-A9	1000	16–64 KB, I/D	0–8 MB
2011	Cortex-A7	1500	32 KB, I/D	0–4 MB
2011	Cortex-A15	2000	32 KB, I/D	0–4 MB
2012	Cortex-M0+	60–250	None	None
2012	Cortex-A53	1500	8–64 KB, I/D	128 KB–2 MB
2012	Cortex-A57	2000	48 KB I / 32 KB D	512 KB–2 MB

of transistors. The increasing difference between CPU and memory speeds necessitates a lower miss rate to avoid the main memory bottleneck, and the decreasing cost of transistors allows larger cache sizes.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid state storage as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard drive made of magnetic storage, also called a *hard disk*, with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head

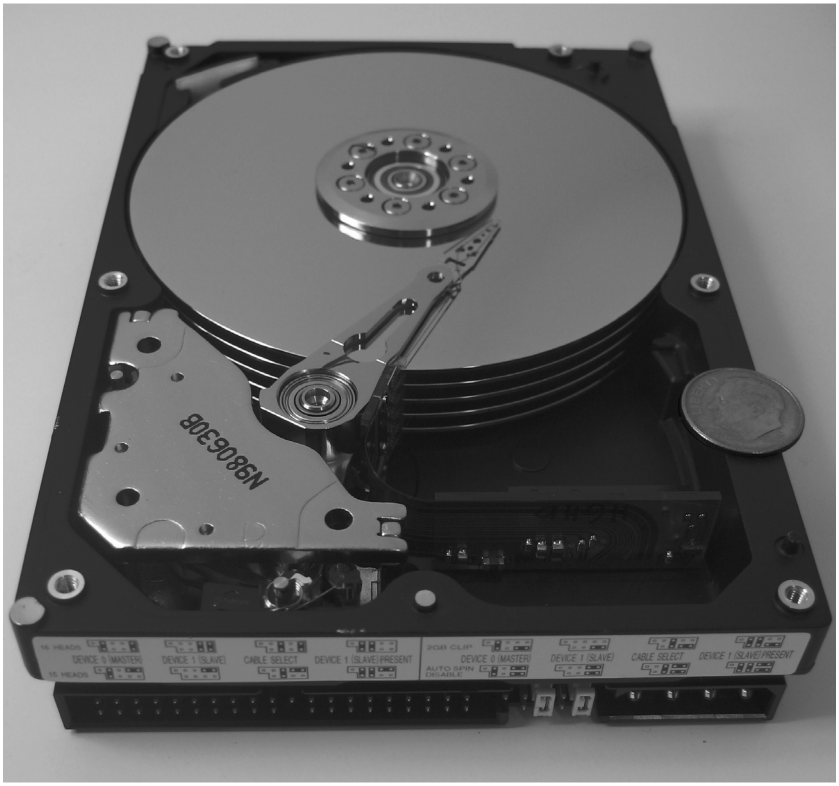

N9806308
16 HEADS
DEVICE 0 (MASTER)
DEVICE 1 (SLAVE)
CABLE SELECT
DEVICE 1 (SLAVE) PRESENT
15 HEADS
2GB CLIP
DEVICE 0 (MASTER)
DEVICE 1 (SLAVE)
CABLE SELECT
DEVICE 1 (SLAVE) PRESENT
AUTO SPIN DISABLE

Figure 8.19 Hard disk

moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor. Hard disk drives are increasingly being replaced by solid state drives because reading is orders of magnitude faster (see Figure 8.4) and they are not as susceptible to mechanical failures.

The objective of adding a hard drive to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard drive. This larger 2-GB memory is called *virtual memory*, and the smaller 128-MB main memory is called *physical memory*. We will use the term physical memory to refer to main memory throughout this section.

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.4 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the hard drive. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the hard drive. The process of determining the physical address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault*

Table 8.4 **Analogous cache and virtual memory terms**

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

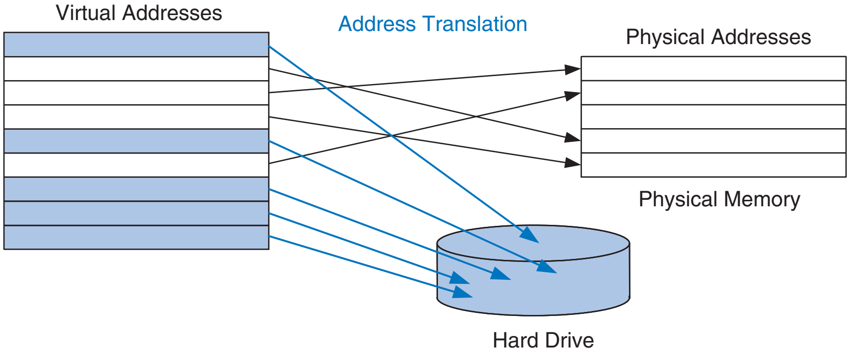


Figure 8.20 **Virtual and physical pages**

occurs, and the operating system loads the page from the hard drive into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access, and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual

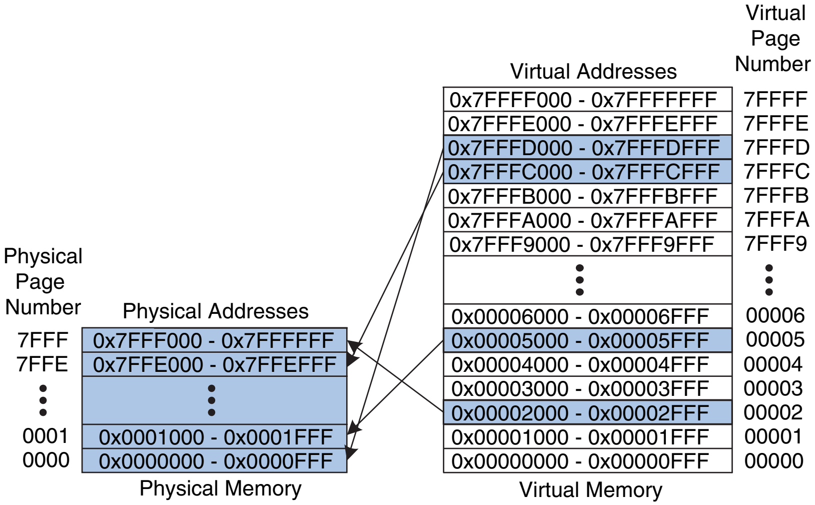

Virtual Page Number
Virtual Addresses
0x7FFFF000 - 0x7FFFFFFF 7FFFF
0x7FFFE000 - 0x7FFFEFFF 7FFFE
0x7FFFD000 - 0x7FFFDFFF 7FFFD
0x7FFFC000 - 0x7FFFCFFF 7FFFC
0x7FFFB000 - 0x7FFFBFFF 7FFFB
0x7FFFA000 - 0x7FFFAFFF 7FFFA
0x7FFF9000 - 0x7FFF9FFF 7FFF9
0x00006000 - 0x00006FFF 00006
0x00005000 - 0x00005FFF 00005
0x00004000 - 0x00004FFF 00004
0x00003000 - 0x00003FFF 00003
0x00002000 - 0x00002FFF 00002
0x00001000 - 0x00001FFF 00001
0x00000000 - 0x00000FFF 00000
Virtual Memory
Physical Page Number
Physical Addresses
7FFF 0x7FFF000 - 0x7FFFFFF
7FFE 0x7FFE000 - 0x7FFEFFF
0001 0x0001000 - 0x0001FFF
0000 0x0000000 - 0x0000FFF
Physical Memory

Figure 8.21 **Physical and virtual pages**

addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

Because the page size is 4 KB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

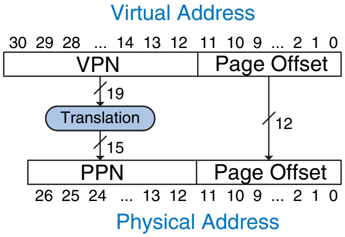


Figure 8.22 **Translation from virtual address to physical address**

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution: The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on the hard drive.

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Figure 8.23 **The page table for Figure 8.21**

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution: Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

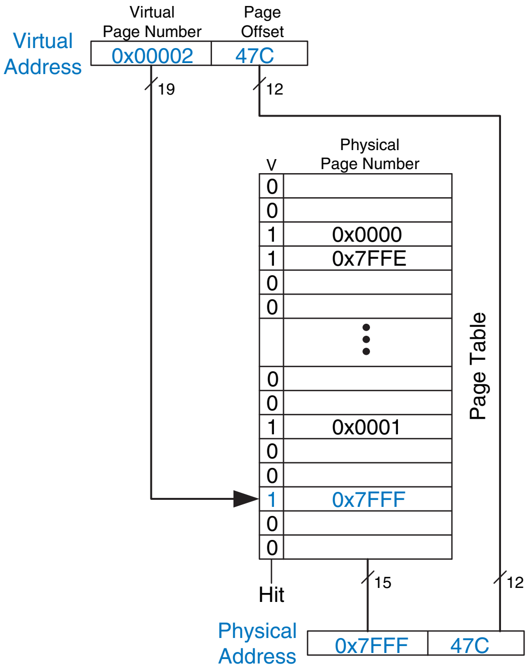

Virtual Address
Virtual Page Number
Page Offset
0x00002
47C
19
12
V
Physical Page Number
0
0
1 0x0000
1 0x7FFE
0
0
0
0
1 0x0001
0
0
1 0x7FFF
0
0
Page Table
15
12
Hit
Physical Address
0x7FFF
47C

Figure 8.24 Address translation using the page table

The page table can be stored anywhere in physical memory, at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFFD.

Solution: Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

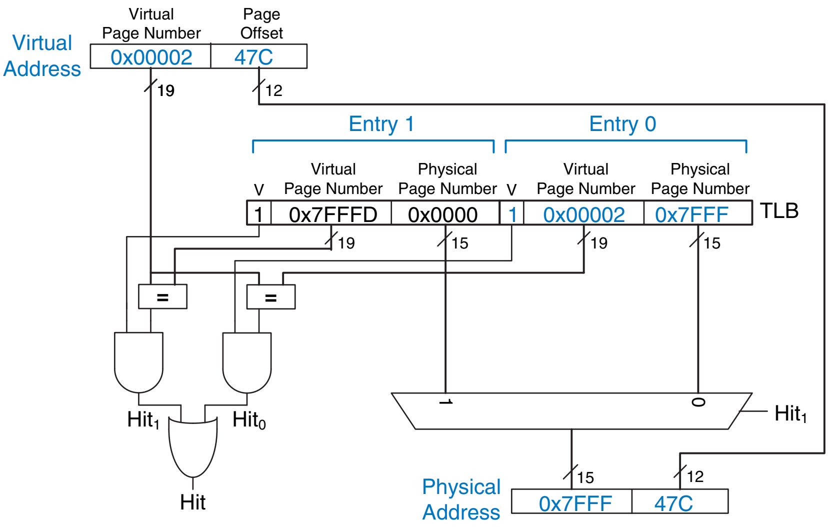

Virtual Address
Virtual Page Number
Page Offset
0x00002
47C
19
12
Entry 1
Entry 0
V
Virtual Page Number
Physical Page Number
V
Virtual Page Number
Physical Page Number
1
0x7FFFD
0x0000
1
0x00002
0x7FFF
TLB
19
15
19
15
=
=
Hit1
Hit0
Hit
1
0
Hit1
15
12
Physical Address
0x7FFF
47C

Figure 8.25 **Address translation using a two-entry TLB**

8.4.4 Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages, because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The operating system adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be impractical. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called *paging*, and the hard drive in a virtual memory system is sometimes called *swap space*. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from the hard drive. When a physical page is paged out, it needs to be written back to the hard drive only if its dirty bit is 1; otherwise, the hard drive already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to page out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GB virtual memory with 4 KB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^{2}$ bytes $= 2^{21}$ bytes $= 2$ MB.

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be paged out to the hard drive so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that it must be fetched from the

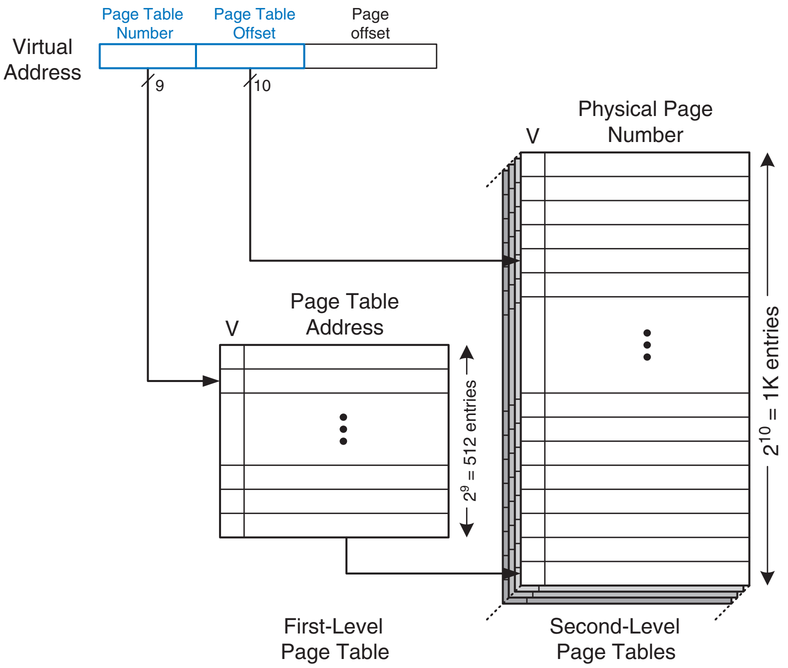

Virtual Address
Page Table Number
Page Table Offset
Page offset
9
10
Physical Page Number
V
Page Table Address
V
2^9 = 512 entries
2^{10} = 1K entries
First-Level Page Table
Second-Level Page Tables

Figure 8.26 **Hierarchical page tables**

hard drive when *V* is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KB.

In Figure 8.26 the 19-bit virtual page number is broken into 9 and 10 bits, to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ K entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the hierarchical page table uses only $(512 \times 4 \text{ bytes}) + 2 \times (1\text{ K} \times 4 \text{ bytes}) =$ 10 KB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution: As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

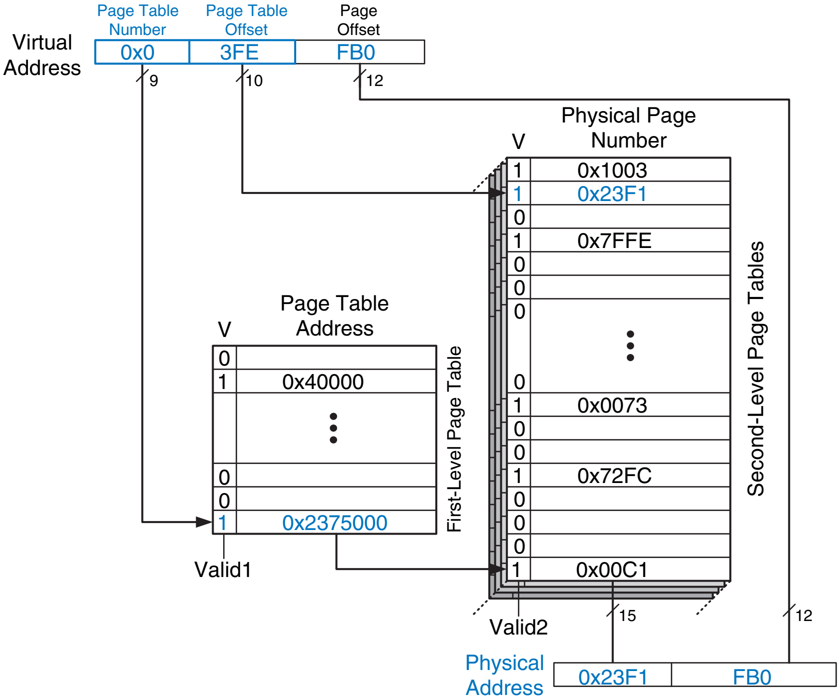

Virtual Address
Page Table Number
Page Table Offset
Page Offset
0x0
3FE
FB0
9
10
12
Page Table Address
V
0
1
0x40000
0
0
1
0x2375000
Valid1
First-Level Page Table
Physical Page Number
V
1
0x1003
1
0x23F1
0
1
0x7FFE
0
0
0
0
1
0x0073
0
0
1
0x72FC
0
0
0
1
0x00C1
Second-Level Page Tables
Valid2
15
12
Physical Address
0x23F1
FB0

Figure 8.27 Address translation using a two-level page table

8.5 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies, such as DRAM, SRAM, and hard drives, offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory.

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program an ARM processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. Hackers and governments wage undeclared cyberwarfare, attacking industrial infrastructure and financial networks. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality, and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16-KB direct mapped cache always has a lower miss rate than an 8-KB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, C?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are S and N for a fully associative cache of capacity C words with block size b?

(d) What is S for a direct mapped cache of size C words and block size b?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `LDR` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring startup effects (i.e., compulsory misses).

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 1$ word

(c) two-way set associative cache, $b = 1$ word

(d) direct mapped cache, $b = 2$ words

Exercise 8.10 Repeat Exercise 8.9 for the following repeating sequence of `LDR` addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 2$ words

(c) two-way set associative cache, $b = 2$ words

(d) direct mapped cache, $b = 4$ words

Exercise 8.11 Suppose you are running a program with the following data access pattern. The pattern is executed only once.

0x0 0x8 0x10 0x18 0x20 0x28

(a) If you use a direct mapped cache with a cache size of 1 KB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

(i) Increasing the degree of associativity to 2.

(ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.12 You are building an instruction cache for an ARM processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.13 Consider a cache with the following parameters: N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits, C (cache size) = 32 K words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.28 and a small number of two-input logic gates. The cache design must include tag storage, data

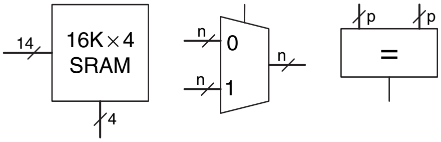


Figure 8.28 **Building blocks**

storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (*n* or *p* bits wide, respectively), but the SRAM blocks must be 16K × 4 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

Exercise 8.14 You've joined a hot new Internet startup to build wrist watches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.29. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

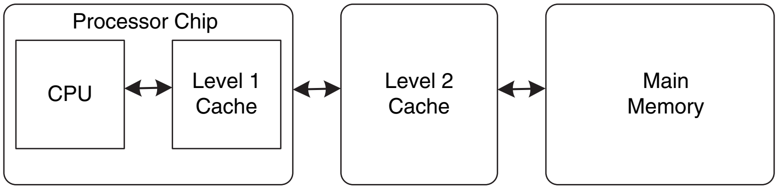


Figure 8.29 **Computer system**

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.5. The DRAM has an access time of t_m and a size of 512 MB.

Table 8.5 **Memory characteristics**

Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256K

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.15 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other, less common, replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.16 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the ARM multicycle processor from Figure 7.30 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.5 that has 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.17 Repeat Exercise 8.16 with the following parameters.

(a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.5 that has 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.18 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte*, 2^{50} bytes = 1 *petabyte*, and 2^{60} bytes = 1 *exabyte*.

Exercise 8.19 A supercomputer designer chooses to spend $1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

Exercise 8.20 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space, but are limited to only 8 MB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this “direct mapping” a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (e). Suppose you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (V) and a dirty bit (D). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.21 Consider a virtual memory system that can address a total of 2^{50} bytes. You have unlimited hard drive space, but are limited to 2 GB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) How many page table entries will the page table contain?

(f) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (V) and a dirty bit (D). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(g) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.22 You decide to speed up the virtual memory system of Exercise 8.20 by using a translation lookaside buffer (TLB). Suppose your memory system has the characteristics shown in Table 8.6. The TLB and cache miss rates indicate how

Table 8.6 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
Cache	1	2%
Main memory	100	0.0003%
Hard drive	1,000,000	0%

often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.23 You decide to speed up the virtual memory system of Exercise 8.21 by using a translation lookaside buffer (TLB) with 128 entries.

(a) How big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(b) Sketch the TLB. Clearly label all fields and dimensions.

(c) What size SRAM would you need to build the TLB described in part (b)? Give your answer in terms of depth × width.

Exercise 8.24 Suppose the ARM multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.25 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KB) pages. Each page table entry contains a physical page number, a valid bit (V), and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.26 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.27 Suppose you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard drive affect performance?

(c) How does the size of your PC's physical memory affect performance?

Interview Questions

The following exercises present questions that have been asked on interviews.

Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.

This page intentionally left blank

9

I/O Systems

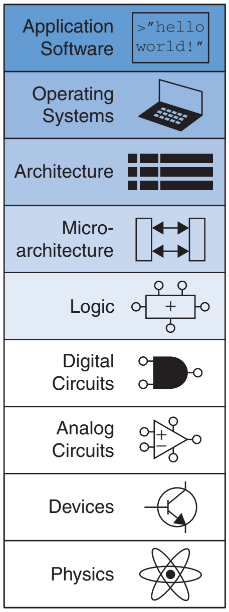
9.1 Introduction
9.2 Memory-Mapped I/O
9.3 Embedded I/O Systems
9.4 Other Microcontroller Peripherals
9.5 Bus Interfaces
9.6 PC I/O Systems
9.7 Summary

9.1 INTRODUCTION

Input/Output (I/O) systems are used to connect a computer with external devices called peripherals. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems, showing how to use an ARM-based Raspberry Pi single-board computer to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers. Section 9.4 gives examples of interfacing with other common devices such as character LCDs, VGA monitors, Bluetooth radios, and motors. Section 9.5 describes bus interfaces and illustrates the popular AHB-Lite bus. Section 9.6 surveys the major I/O systems used in PCs.

The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.



Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00009-1
© 2016 Elsevier Inc. All rights reserved.

I/O Systems

e9

9.1 Introduction
9.2 Memory-Mapped I/O
9.3 Embedded I/O Systems
9.4 Other Microcontroller Peripherals
9.5 Bus Interfaces
9.6 PC I/O Systems
9.7 Summary

9.1 INTRODUCTION

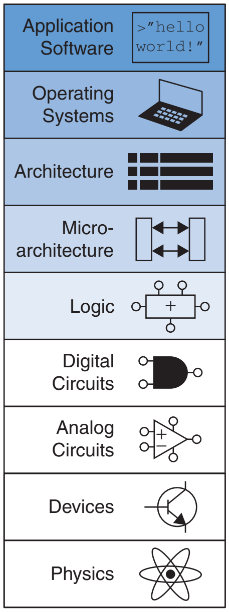
Input/Output (I/O) systems are used to connect a computer with external devices called peripherals. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems, showing how to use an ARM-based Raspberry Pi single-board computer to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers. Section 9.4 gives examples of interfacing with other common devices such as character LCDs, VGA monitors, Bluetooth radios, and motors. Section 9.5 describes bus interfaces and illustrates the popular AHB-Lite bus. Section 9.6 surveys the major I/O systems used in PCs.

9.2 MEMORY-MAPPED I/O

Recall from Section 6.5.1 that a portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that physical addresses in the range 0x20000000 to 0x20FFFFFF are used for I/O. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped* I/O.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure e9.1 shows the hardware needed to support two memory-mapped I/O devices. An address decoder determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of the hardware. The *ReadData* multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.



Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00015-7
© 2016 Elsevier Inc. All rights reserved.

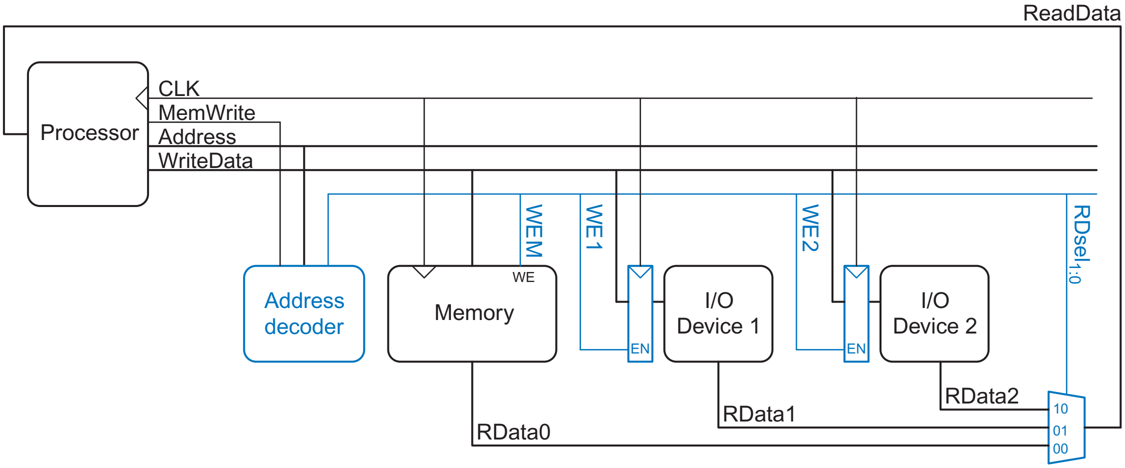

ReadData
CLK
MemWrite
Address
WriteData
Processor
Address decoder
Memory
WE
WEM
WE1
WE2
EN
EN
I/O Device 1
I/O Device 2
RDsel1:0
RData0
RData1
RData2
10
01
00

Figure e9.1 **Support hardware for memory-mapped I/O**

Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following assembly code writes the value 7 to I/O Device 1.

```
        MOV R1, #7
        LDR R2, =ioadr
        STR R1, [R2]
ioadr DCD 0x20001000
```

The address decoder asserts *WE1* because the address is 0x20001000 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following assembly code.

```
   LDR R1, [R2]
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0x20001000 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into R1 in the processor.

The addresses associated with I/O devices are often called *I/O registers* because they may correspond with physical registers in the I/O device like those shown in Figure e9.1.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware including the addresses and behavior of the memory-mapped I/O registers. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

9.3 EMBEDDED I/O SYSTEMS

Embedded systems use a processor to control interactions with the physical environment. They are typically built around microcontroller units (MCUs) which combine a microprocessor with a set of easy-to-use peripherals such as general-purpose digital and analog I/O pins, serial ports, timers, etc. Microcontrollers are generally inexpensive and are designed to minimize system cost and size by integrating most of the necessary components onto a single chip. Most are smaller and lighter than a dime, consume milliwatts of power, and range in cost from a few dimes up to several dollars. Microcontrollers are classified by the size of data that they operate upon. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

For the sake of concreteness, this section will illustrate embedded system I/O in the context of a real system. Specifically, we will focus on the popular and inexpensive Raspberry Pi board, which contains a Broadcom BCM2835 system-on-chip (SoC) with a 700 MHz 32-bit ARM1176JZ-F processor implementing the ARMv6 instruction set. The principles in each subsection will be followed by specific examples that run on the Pi. All of the examples have been tested on a Pi running NOOBS Raspbian Linux in 2014.

Figure e9.2 shows a photograph of a Raspberry Pi Model B+ board, which is a complete Linux computer about the size of a credit card that sells for $35. The Pi draws up to 1 A from a 5 V USB power supply. It has 512 MB of onboard RAM and an SD card socket for a memory card that contains the operating system and user files. Connectors provide video and audio output, USB ports for a mouse and keyboard, and an Ethernet (Local Area Network) port, along with 40 general-purpose I/O (GPIO) pins that are the main subject of this chapter.

While the BCM2835 SoC has many capabilities beyond those in a typical inexpensive microcontroller, the general-purpose I/O is very similar. This chapter begins by describing the BCM2835 on the Raspberry Pi and describing a device driver for memory-mapped I/O. The remainder of this chapter will illustrate how embedded systems perform general-purpose digital, analog, and serial I/O. Timers are also commonly used to generate or measure precise time intervals.

Some architectures, notably x86, use specialized instructions instead of memory-mapped I/O to communicate with I/O devices. These instructions are of the following form, where `device1` and `device2` are the unique IDs of the peripheral device:

```
LDRIO R1, device1
STRIO R2, device2
```

This type of communication with I/O devices is called *programmed I/O*.

Approximately $19B of microcontrollers were sold in 2014, and the market is forecast to reach $27B by 2020. The average price of a microcontroller is less than $1, and an 8-bit microcontroller can be integrated on a system-on-chip for less than a penny. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 150 in each home and 50 in each automobile in 2010. The 8051 is a classic 8-bit microcontroller originally developed by Intel in 1980 and now sold by a host of manufacturers. Microchip's PIC16 and PIC18-series are 8-bit market leaders. The Atmel AVR series of microcontrollers has been popularized among hobbyists as the brain of the Arduino platform. Among 32-bit microcontrollers, Renesas leads the overall market. Freescale, Samsung, Texas Instruments, and Infineon are other major microcontroller manufacturers. ARM processors are found in nearly all smart phones and tablets today and are usually part of a system-on-chip containing the multi-core applications processor, a graphics processing unit, and extensive I/O.

The Raspberry Pi was developed in 2011-12 by the nonprofit Raspberry Pi Foundation in the UK to promote teaching computer science. Built around the brain of an inexpensive smartphone, the computer has become wildly popular, selling more than 3 million units by 2014. The name pays homage to early home computers including Apple, Apricot, and Tangerine. Pi is derived from Python, a programming language often used in education. Documentation and purchasing information can be found at

`raspberrypi.org`

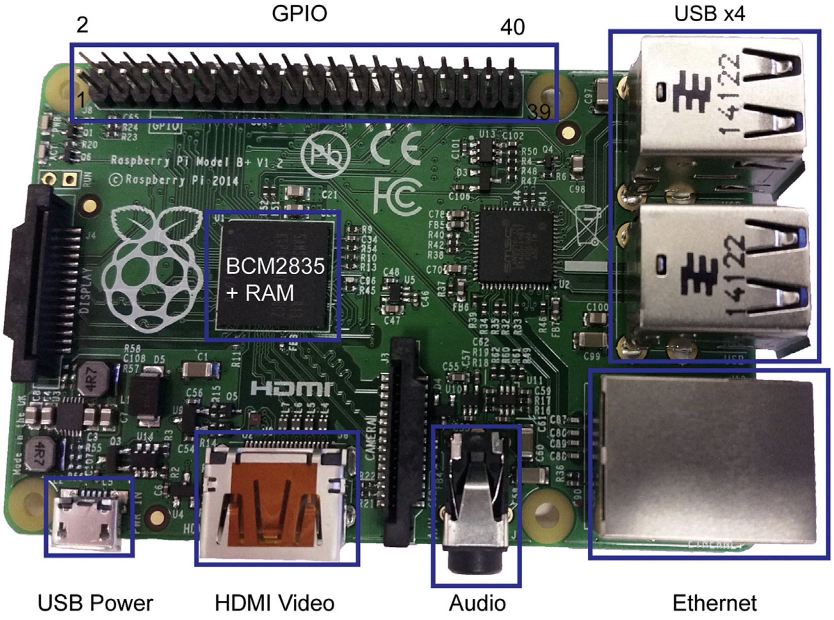


Figure e9.2 Raspberry Pi Model B+

Eben Upton (1978-) is the architect of the Raspberry Pi and a founder of the Raspberry Pi Foundation. He received his Bachelor's and Ph.D. from the University of Cambridge before joining Broadcom Corporation as a chip architect.

(Photograph © Eben Upton. Reproduced with permission.)

9.3.1 BCM2835 System-on-Chip

The BCM2835 SoC is a powerful yet inexpensive chip designed by Broadcom for mobile devices and other multimedia applications. The SoC includes an ARM microprocessor known as the *applications processor*, a VideoCore processor for graphics, video, and cameras, and many I/O peripherals. The BCM2835 is packaged in a plastic ball grid array with tiny solder balls underneath; it is best soldered by a robot that aligns the package to matching copper pads on a printed circuit board and applies heat. Broadcom does not publish a complete datasheet, but an abbreviated datasheet is available on the Raspberry Pi site describing how to access peripherals from the ARM processor. The datasheet describes many features and I/O registers that are omitted in this chapter for simplicity.

`www.raspberrypi.org/documentation/hardware/`

Figure e9.3 shows a simplified schematic of the Raspberry Pi model board. The board receives 5 V power from a USB power supply and regulators produce 3.3, 2.5, and 1.8 V levels for I/O, analog, and

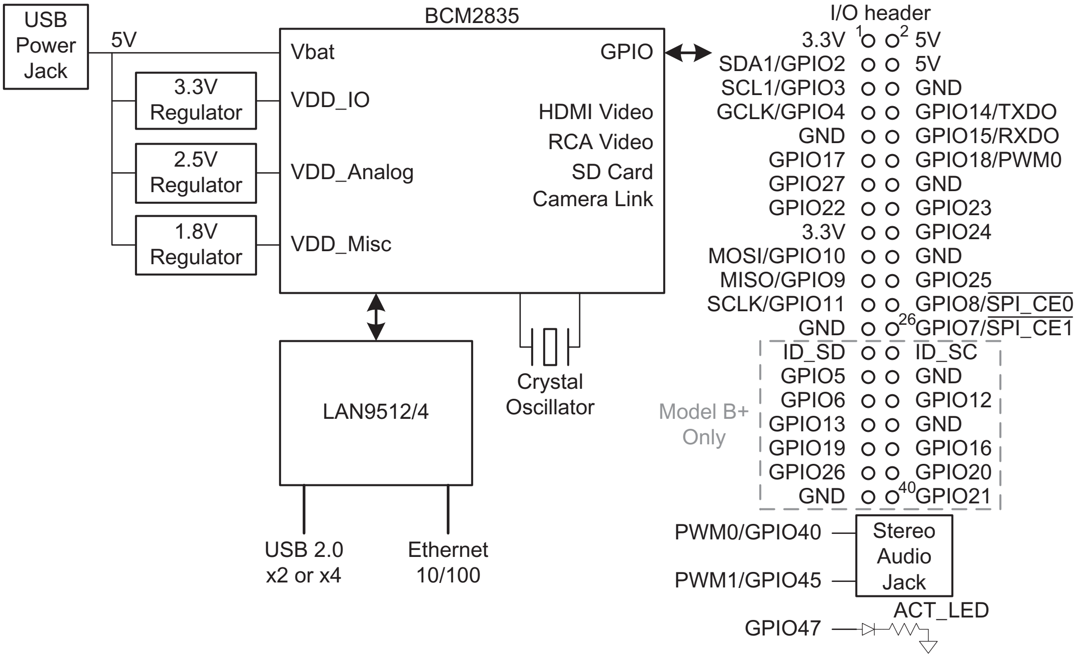

USB Power Jack
5V
3.3V Regulator
2.5V Regulator
1.8V Regulator
BCM2835
Vbat
VDD_IO
VDD_Analog
VDD_Misc
GPIO
HDMI Video
RCA Video
SD Card
Camera Link
LAN9512/4
USB 2.0 x2 or x4
Ethernet 10/100
Crystal Oscillator
I/O header
3.3V 1 2 5V
SDA1/GPIO2 5V
SCL1/GPIO3 GND
GCLK/GPIO4 GPIO14/TXDO
GND GPIO15/RXDO
GPIO17 GPIO18/PWM0
GPIO27 GND
GPIO22 GPIO23
3.3V GPIO24
MOSI/GPIO10 GND
MISO/GPIO9 GPIO25
SCLK/GPIO11 GPIO8/SPI_CE0
GND 26 GPIO7/SPI_CE1
ID_SD ID_SC
GPIO5 GND
GPIO6 GPIO12
GPIO13 GND
GPIO19 GPIO16
GPIO26 GPIO20
GND 40 GPIO21
Model B+ Only
PWM0/GPIO40
PWM1/GPIO45
Stereo Audio Jack
ACT_LED
GPIO47

Figure e9.3 **Raspberry Pi I/O schematic**

miscellaneous functions. The BCM2835 also has an internal switching regulator that produces a variable lower voltage for the power-efficient SoC. The BCM2835 connects to a USB/Ethernet controller and also directly outputs video. It also has 54 configurable I/O signals but, for space reasons, only a fraction of these are accessible to the user via header pins. The header also provides 3.3 and 5 V and ground to conveniently power small devices attached to the Pi, but the maximum total current is 50 mA from 3.3 V and ~300 mA from 5 V. The model B and B+ are similar, but B+ boosts the number of I/O header pins from 26 to 40 and the number of USB ports from 2 to 4. Various cables including the Adafruit Pi Cobbler are available to connect these header pins to a breadboard.

The Raspberry Pi uses an SD card as a Flash memory disk. The card is typically preloaded with Raspbian Linux, a small version of Linux that fits on an 8 GB SD card. You can work with the Pi either by attaching an HDMI monitor and USB mouse and keyboard to turn it into a full computer, or by connecting to it from another computer over an Ethernet cable.

The Raspberry Pi continues to advance and by the time you read this, a newer model might be available with a more advanced processor and a different set of embedded I/O. Nevertheless, the same principles will apply, and the principles also apply to other types of microcontrollers. You can expect to find the same types of I/O peripherals. You will need to consult the data sheet to look up the mapping between the peripheral, the pin on the chip, and the pin on the board, as well as the addresses of the memory-mapped I/O registers associated with each peripheral. You'll write configuration registers to initialize the peripheral and read and write data registers to access the peripheral.

9.3.2 Device Drivers

Programmers can manipulate I/O devices directly by reading or writing the memory-mapped I/O registers. However, it is better programming practice to call functions that access the memory-mapped I/O. These functions are called *device drivers*. Some of the benefits of using device drivers include:

- The code is easier to read when it involves a clearly named function call rather than a write to bit fields at an obscure memory address.
- Somebody who is familiar with the deep workings of the I/O devices can write the device driver and casual users can call it without having to understand the details.
- The code is easier to port to another processor with different memory mapping or I/O devices because only the device driver must change.
- If the device driver is part of the operating system, the OS can control access to physical devices shared among multiple programs running on the system and can manage security (e.g. so a malicious program can't read the keyboard while you are typing your password into a web browser).

This section will develop a simple device driver called EasyPIO to access BCM2835 devices so that you can understand what is happening under the hood in a device driver. Casual users are likely to prefer WiringPi, an open-source I/O library for the Pi, which has functions similar to but not exactly matching those in EasyPIO.

The memory-mapped I/O on the BCM2835 is found at physical addresses 0x20000000-0x20FFFFFF. The physical base addresses used by various peripherals are summarized in Table e9.1. Peripherals have multiple I/O registers starting at their base address. For example, reading address 0x20200034 will return the values of GPIO (general-purpose I/O) pins 31:0. The peripherals in bold will be discussed further in subsequent sections.

The Raspberry Pi typically runs a Linux operating system using virtual memory, which further complicates memory-mapped I/O. Loads and stores in a program refer to virtual addresses, not physical, so a program cannot immediately access memory-mapped I/O. Instead, it must begin by asking the operating system to map the physical addresses of interest to the program's virtual address space. The `pioInit` function from EasyPIO in Example e9.2 performs this task. The code involves some heavy duty pointer manipulation in C. The general principle is to open `/dev/mem`, which is a Linux method of accessing physical memory. Then the `mmap` function is used to set `gpio` as a pointer to physical address 0x20200000, the beginning of the GPIO registers. The pointer is declared

As this book was going to press, the Raspberry Pi Foundation released the Raspberry Pi 2 Model B with a BCM2836 SoC containing a quad Cortex-A7 processor and 1 GB of RAM. The Pi 2 runs about 6 times faster than the B+ but has the same I/O as the B+ described in this chapter. The peripheral base address has moved from 0x20000000 to 0x3F000000. An updated EasyPIO supporting both models is posted to the textbook website.

Caution: the I/O connector pinout has changed between Raspberry Pi board revisions.

Caution: connecting 5 V to one of the 3.3 V I/Os will damage the I/O and possibly the entire Raspberry Pi. If you probe the I/O pins with a voltmeter, beware that you do not accidentally make contact between the 5 V pins and a nearby pin!

Pin 1 of the I/O header is labeled in Figure e9.3. When you make connections, be sure you have properly identified it and aren't rotated by 180 degrees. This is an easy mistake that could cause you to accidentally damage the Pi.

EasyPIO and the code examples in this chapter can be downloaded from the textbook website: http://booksite.elsevier.com/9780128000564. The WiringPi driver and documentation is at `wiringpi.com`.

Table e9.1 **Memory mapped I/O addresses**

Physical Base Address	Peripheral
0x20003000	**System Timer**
0x2000B200	Interrupts
0x2000B400	ARM Timer
0x20200000	**GPIO**
0x20201000	**UART0**
0x20203000	PCM Audio
0x20204000	**SPI0**
0x20205000	I^{2}C Master #1
0x2020C000	**PWM**
0x20214000	I^{2}C Slave
0x20215000	miniUART1, SPI1, SPI2
0x20300000	SD Card Controller
0x20804000	I^{2}C Master #2
0x20805000	I^{2}C Master #3

`volatile`, telling the compiler that the memory-mapped I/O value might change on its own, so the program should always read the register directly instead of relying on an old value. GPLEV0 accesses the I/O register 13 words past GPIO, e.g. at 0x20200034, which contains the values of GPIO 31:0. For brevity, this example omits error checking that takes place in the actual EasyPIO library. Subsequent subsections define more registers and functions to access I/O devices.

For security reasons, Linux only grants the *superuser* access to memory-mapped hardware. To run a program as the superuser, type `sudo` before the Linux command. The next section will give an example.

Example e9.2 INITIALIZING MEMORY-MAPPED I/O

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE   0x20000000
#define GPIO_BASE           (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int       *gpio; //Pointer to base of gpio
#define GPLEV0              (* (volatile unsigned int *) (gpio + 13))
#define BLOCK_SIZE          (4*1024)
```

```
void pioInit(){
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in Linux
    mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
    reg_map = mmap(
      NULL,                   // Address at which to start local mapping (null = don't-care)
      BLOCK_SIZE,             // 4KB mapped memory block
      PROT_READ|PROT_WRITE,   // Enable both reading and writing to the mapped memory
      MAP_SHARED,             // Nonexclusive access to this memory
      mem_fd,                 // Map to /dev/mem
      GPIO_BASE);             // Offset to GPIO peripheral

    gpio = (volatile unsigned *)reg_map;
    close(mem_fd);
}
```

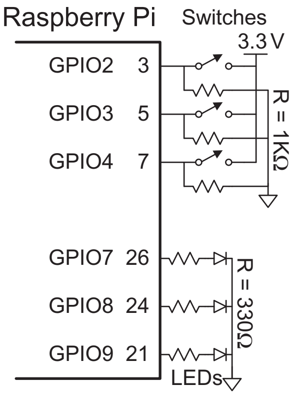


Figure e9.4 LEDs and switches connected to GPIO pins

In the context of bit manipulation, "setting" means writing to 1 and "clearing" means writing to 0.

9.3.3 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. For example, Figure e9.4 shows three light-emitting diodes (LEDs) and three switches connected to six GPIO pins. The LEDs are wired to glow when driven with a 1 and to turn off when driven with a 0. The current-limiting resistors are placed in series with the LEDs to set the brightness and avoid overloading the current capability of the GPIO. The switches are wired to produce a 1 when closed and a 0 when open. The schematic indicates the pin name as well as the corresponding header pin number.

At a minimum, any GPIO pin requires registers to read input pin values, write output pin values, and set the direction of the pin. In many embedded systems, the GPIO pins can be shared with one or more special-purpose peripherals, so additional configuration registers are necessary to determine whether the pin is general or special-purpose. Furthermore, the processor may generate interrupts when an event such as a rising or falling edge occurs on an input pin, and configuration registers may be used to specify the conditions for an interrupt.

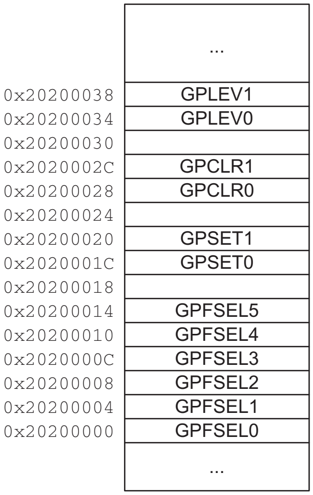
Recall that the BCM2835 has 54 GPIOs. They are controlled by the GPFSEL, GPLEV, GPSET, and GPCLR registers. Figure e9.5 shows a memory map for these GPIO registers. GPFSEL5…0 determine whether each pin is a general-purpose input, output, or special-purpose I/O. Each of these function select registers uses 3 bits to specify each pin and thus

Table e9.2 **GPFSEL register bit field to GPIO mapping**

	GPFSEL0	GPFSEL1	GPFSEL2	GPFSEL3	GPFSEL4	GPFSEL5
[2:0]	GPIO0	GPIO10	GPIO20	GPIO30	GPIO40	GPIO50
[5:3]	GPIO1	GPIO11	GPIO21	GPIO31	GPIO41	GPIO51
[8:6]	GPIO2	GPIO12	GPIO22	GPIO32	GPIO42	GPIO52
[11:9]	GPIO3	GPIO13	GPIO23	GPIO33	GPIO43	GPIO53
[14:12]	GPIO4	GPIO14	GPIO24	GPIO34	GPIO44	
[17:15]	GPIO5	GPIO15	GPIO25	GPIO35	GPIO45	
[20:18]	GPIO6	GPIO16	GPIO26	GPIO36	GPIO46	
[23:21]	GPIO7	GPIO17	GPIO27	GPIO37	GPIO47	
[26:24]	GPIO8	GPIO18	GPIO28	GPIO38	GPIO48	
[29:27]	GPIO9	GPIO19	GPIO29	GPIO39	GPIO49	

each 32-bit register controls 10 GPIOs as given in Table e9.2 and six GPFSEL registers are necessary to control all 54 GPIOs. For example, GPIO13 is configured by GPFSEL1[11:9]. The configurations are summarized in Table e9.3; many pins have multiple special-purpose functions that will be discussed in subsequent sections; ALT0 is most commonly used. Reading GPLEV1…0 returns the values of the pins. For example, GPIO14 is read as GPLEV0[14] and GPIO34 is read as GPLEV1[2]. The pins cannot be directly written; instead, bits are forced high or low by asserting the corresponding bit of GPSET1…0 or GPCLR1…0. For example, GPIO14 is forced to 1 by writing GPSET0[14] = 1 and forced to 0 by writing GPCLR0[14] = 1.

The BCM2835 datasheet does not specify the logic levels or output current capability of the GPIOs. However, users have determined empirically that one should not try to draw more than 16 mA from any single I/O or 50 mA total from all the I/Os. Thus, a GPIO pin is suitable for driving a small LED but not a motor. The I/Os are generally compatible with other 3.3 V chips but are not 5 V-tolerant.



	...
0x20200038	GPLEV1
0x20200034	GPLEV0
0x20200030	
0x2020002C	GPCLR1
0x20200028	GPCLR0
0x20200024	
0x20200020	GPSET1
0x2020001C	GPSET0
0x20200018	
0x20200014	GPFSEL5
0x20200010	GPFSEL4
0x2020000C	GPFSEL3
0x20200008	GPFSEL2
0x20200004	GPFSEL1
0x20200000	GPFSEL0
	...

Figure e9.5 **GPIO memory map**

Table e9.3 **GPFSEL configuration**

GPFSEL	Pin Function
000	Input
001	Output
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

The BCM2835 has unusually complex GPIO access. Some microcontrollers use a single register to configure whether each pin is input or output and another register to read and write the pins.

Example e9.3 GPIO FOR SWITCHES AND LEDS

Enhance EasyPIO with `pinMode`, `digitalRead`, and `digitalWrite` functions to configure a pin's direction and read or write it. Write a C program using these functions to read the three switches and turn on the corresponding LEDs using the hardware in Figure e9.4.

Solution: The additional EasyPIO code is given below. Because multiple registers are used to control the I/O, the functions must compute which register to access and what bit offset to use within the register. `pinMode` then clears the 0 bits and sets the 1 bits for the intended 3-bit function. `digitalWrite` handles writing either 1 or 0 by using GPSET or GPCLR. `digitalRead` pulls out the value of the desired pin and masks off the others.

```
#define GPFSEL  ((volatile unsigned int *) (gpio + 0))
#define GPSET   ((volatile unsigned int *) (gpio + 7))
#define GPCLR   ((volatile unsigned int *) (gpio + 10))
#define GPLEV   ((volatile unsigned int *) (gpio + 13))
#define INPUT   0
#define OUTPUT  1
...
void pinMode(int pin, int function) {
    int reg      =  pin/10;
    int offset   = (pin%10)*3;
    GPFSEL[reg] &= ~((0b111 & ~function) << offset);
    GPFSEL[reg] |=  ((0b111 &  function) << offset);
}

void digitalWrite(int pin, int val) {
    int reg    = pin / 32;
    int offset = pin % 32;

    if  (val)  GPSET[reg] = 1 << offset;
    else       GPCLR[reg] = 1 << offset;
}

int digitalRead(int pin) {
    int reg    = pin / 32;
    int offset = pin % 32;

    return (GPLEV[reg] >> offset) & 0x00000001;
}
```

The program to read switches and write LEDs is given below. It initializes GPIO access, then sets pins 2–4 as inputs for the switches and pins 7–9 as outputs for the LEDs. It then continuously reads the switches and writes their values to the corresponding LEDs.

```
#include "EasyPIO.h"

void main(void) {
    pioInit();
    // Set GPIO 4:2 as inputs
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(4, INPUT);

    // Set GPIO 9:7 as an output
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    while (1) { // Read each switch and write corresponding LED
        digitalWrite(7, digitalRead(2));
        digitalWrite(8, digitalRead(3));
        digitalWrite(9, digitalRead(4));
    }
}
```

Assuming the program is in a file named dip2led.c and that EasyPIO.h is in the same directory, you can compile and run the program using the following commands on the Raspberry Pi command line .`gcc` is the C compiler. Note that `sudo` is required so that the program can access the protected I/O memory. To stop a running program, press Ctrl-C.

```
gcc dip2led.c -o dip2led
sudo ./dip2led
```

9.3.4 Serial I/O

If a microcontroller needs to send more bits than the number of free GPIO pins, it must break the message into multiple smaller transmissions. In each step, it can send either one bit or several bits. The former is called serial I/O and the latter is called parallel I/O. Serial I/O is popular because it uses few wires and is fast enough for many applications. Indeed, it is so popular that many standards for serial I/O have been established and microcontrollers offer dedicated hardware to easily send data via these standards. This section describes the Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (UART) standard serial interfaces.

Other common serial standards include Inter-Integrated Circuit (I^2C), Universal Serial Bus (USB), and Ethernet. I^2C (pronounced "I squared C") is a 2-wire interface with a clock and a bidirectional data pin; it is used in a fashion similar to SPI. USB and Ethernet are more complex, high-performance standards described in Sections 9.6.1 and 9.6.4, respectively. All five of these standards are supported on the Raspberry Pi.

SPI always sends data in both directions on each transfer. If the system only needs unidirectional communication, it can ignore the unwanted data. For example, if the master only needs to send data to the slave, the byte received from the slave can be ignored. If the master only needs to receive data from the slave, it must still trigger the SPI communication by sending an arbitrary byte that the slave will ignore. It can then read the data received from the slave. The SPI clock only toggles while the master is transmitting data.

9.3.4.1 Serial Peripheral Interface (SPI)

SPI (pronounced "S-P-I") is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: Serial Clock (SCK), Master Out Slave In (MOSI, also known as SDO), and Master In Slave Out (MISO, also known as SDI). SPI connects a *master* device to a *slave* device, as shown in Figure e9.6(a). The master produces the clock. It initiates communication by sending a series of clock pulses on SCK. If it wants to send data to the slave, it puts the data on MOSI, starting with the most significant bit. The slave may simultaneously respond by putting data on MISO. Figure e9.6(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable to alert the receiver that data is coming.

The BCM2835 has three SPI master ports and one slave port. This section describes SPI Master Port 0, which is readily accessible on the Raspberry Pi on GPIO pins 11:9. To use these pins for SPI rather than GPIO, their GPFSEL must be set to ALT0. The Pi must then configure the port. When the Pi writes to the SPI, the data is transmitted serially to the slave. Simultaneously, data received from the slave is collected and the Pi can read it when the transfer is complete.

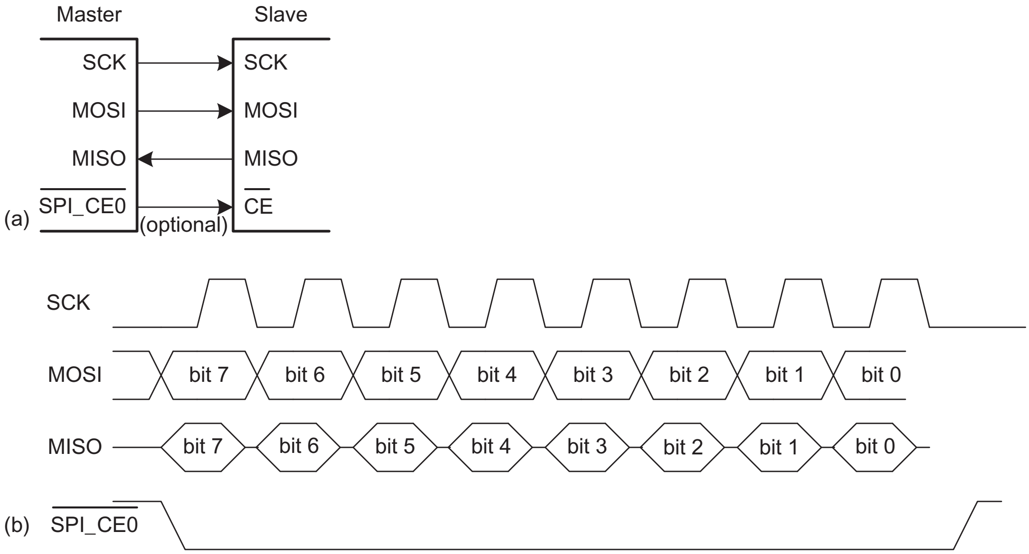


Figure e9.6 SPI connection and waveforms

Table e9.4 **SPI0CS register fields**

Bit	Name	Function	Meaning for 0	Meaning for 1
16	DONE	Transfer Done	Transfer in progress	Transfer complete
7	TA	Transfer Active	SPI disabled	SPI enabled
3	CPOL	Clock Polarity	Clock idles low	Clock idles high
2	CPHA	Clock Phase	First SCK transition at middle of data bit	First SCK transition at beginning of data bit

SPI Master Port 0 is associated with three registers, given in the memory map in Figure e9.7. SPI0CS is the control register. It is used to turn the SPI on and set attributes such as the polarity of the clock. Table e9.4 lists the names and functions of some of the bits in SPI0CS that are relevant to this discussion. All have a default value of 0 on reset. Most of the functions, such as chip selects and interrupts, are not used in this section but can be found in the datasheet. SPI0FIFO is written to transmit a byte and read to get the byte received back. SPI0CLK configures the SPI clock frequency by dividing the 250 MHz peripheral clock by a power of two specified in the register. Thus, the SPI clock frequency is summarized in Table e9.5.

Example e9.4 SENDING AND RECEIVING BYTES OVER SPI

Design a system to communicate between a Raspberry Pi master and an FPGA slave over SPI. Sketch a schematic of the interface. Write the C code for the Pi to send the character ‘A’ and receive a character back. Write HDL code for an SPI slave on the FPGA. How could the slave be simplified if it only needs to receive data?

Solution: Figure e9.8 shows the connection between the devices using SPI Master Port 0. The pin numbers are obtained from the component datasheets (e.g., Figure e9.3). Notice that both the pin numbers and signal names are shown on the diagram to indicate both the physical and logical connectivity. When the SPI is enabled, these pins cannot be used for GPIO.

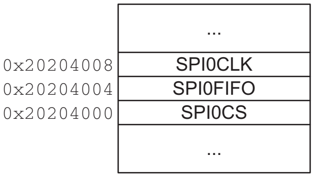


Figure e9.7 **SPI Master Port 0 registers**

If the frequency is too high (>~1 MHz on a breadboard or tens of MHz on an unterminated printed circuit board), the SPI may become unreliable due to reflections, crosstalk, or other signal integrity issues.

Table e9.5 **SPI0CLK frequencies**

SPI0CLK	SPI Frequency (kHz)
2	125000
4	62500
8	31250
16	15625
32	7812
64	3906
128	1953
256	976
512	488
1024	244
2048	122

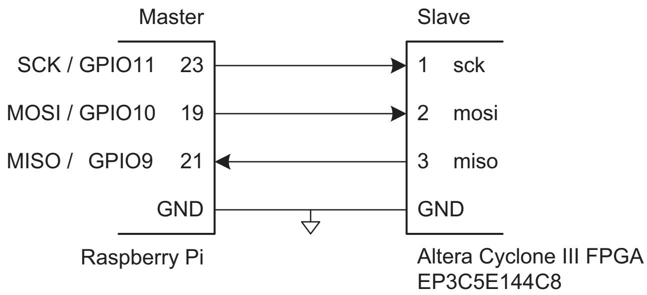


Figure e9.8 **SPI connection between Pi and FPGA**

The following code from EasyPIO.h is used to initialize the SPI and to send and receive a character. The code to set up the memory map and define the register addresses is similar to that for GPIO and is not reprinted here.

```
void spiInit(int freq, int settings) {
    pinMode(8, ALT0);           // CE0b
    pinMode(9, ALT0);           // MISO
    pinMode(10, ALT0);          // MOSI
    pinMode(11, ALT0);          // SCLK

    SPI0CLK = 250000000/freq;  // Set SPI clock divider to desired
                                  freq
    SPI0CS = settings;
    SPI0CSbits.TA = 1;          // Turn SPI on
}

char spiSendReceive(char send){
    SPI0FIFO = send;            // Send data to slave
    while (!SPI0CSbits.DONE);   // Wait until SPI complete
    return SPI0FIFO;            // Return received data
}
```

The C code below initializes the SPI and then sends and receives a character. It sets the SPI clock to 244 kHz.

```
#include "EasyPIO.h"

void main(void) {
    char received;

    pioInit();
    spiInit(244000, 0);              // Initialize the SPI:
                                     // 244 kHz clk, default settings
    received = spiSendReceive('A');  // Send letter A and receive byte
}
```

The HDL code for the FPGA is listed below. Figure e9.9 shows a block diagram and timing. The FPGA uses a shift register to hold the bits that have been received from the master and the bits that remain to be sent to the master.

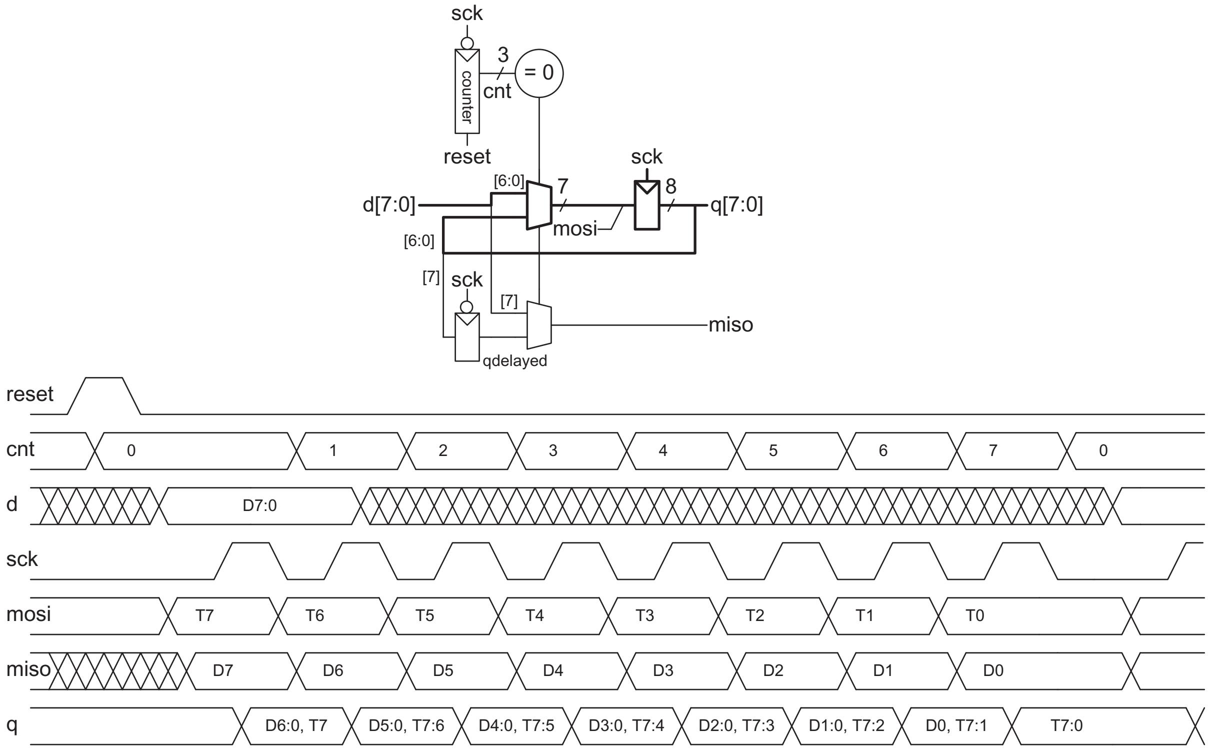

sck
counter
3
cnt
= 0
reset
sck
[6:0]
7
8
d[7:0]
q[7:0]
mosi
[6:0]
[7]
sck
[7]
miso
qdelayed
reset
cnt
0
1
2
3
4
5
6
7
0
d
D7:0
sck
mosi
T7
T6
T5
T4
T3
T2
T1
T0
miso
D7
D6
D5
D4
D3
D2
D1
D0
q
D6:0, T7
D5:0, T7:6
D4:0, T7:5
D3:0, T7:4
D2:0, T7:3
D1:0, T7:2
D0, T7:1
T7:0

Figure e9.9 SPI slave circuitry and timing

On the first rising `sck` edge after reset and each 8 cycles thereafter, a new byte from `d` is loaded into the shift register. On each subsequent cycle, a bit is shifted in on `mosi` and a bit is shifted out of `miso`. `miso` is delayed until the falling edge of `sck` so that it can be sampled by the master on the next rising edge. After 8 cycles, the byte received can be found in `q`.

```
module spi_slave(input  logic       sck,      // From master
                 input  logic       mosi,     // From master
                 output logic       miso,     // To master
                 input  logic       reset,    // System reset
                 input  logic [7:0] d,        // Data to send
                 output logic [7:0] q);       // Data received

  logic [2:0] cnt;
  logic       qdelayed;

  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else       cnt = cnt + 3'b1;
  // Loadable shift register
  // Loads d at the start, shifts mosi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

  // Align miso to falling edge of sck
  // Load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

If the slave only needs to receive data from the master, it reduces to a simple shift register given in the following HDL code.

```
module spi_slave_receive_only(input  logic       sck,  //From master
                              input  logic       mosi,//From master
                              output logic [7:0] q);  //Data received
  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

SPI ports are highly configurable so that they can talk to a wide variety of serial devices. Unfortunately, this leads to the possibility of incorrectly configuring the port and garbling the data transmission. Sometimes it is necessary to change the configuration bits to communicate with a device that expects different timing. When CPOL = 1, SCK is inverted. When CPHA = 1, the clocks toggle half a cycle earlier relative to the data. These modes are shown in Figure e9.10. Be aware that different SPI products may use different names and polarities for these options; check the waveforms carefully for your device. It can also be helpful to examine

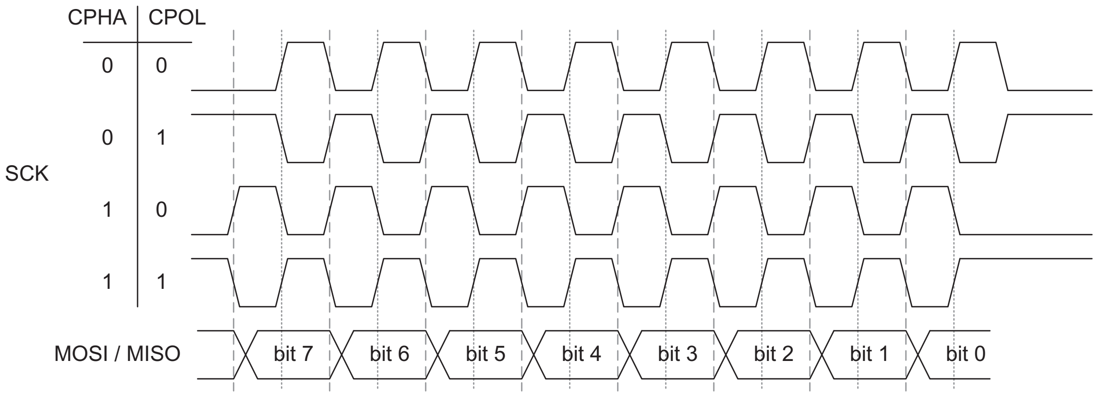


Figure e9.10 SPI clock and data timing configurations

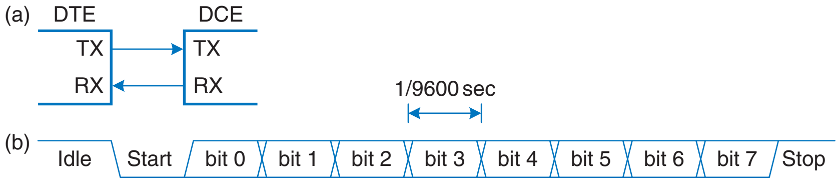


Figure e9.11 **Asynchronous serial link**

SCK, MOSI, and MISO on an oscilloscope if you are having communication difficulties.

9.3.4.2 Universal Asynchronous Receiver/Transmitter (UART)

A UART (pronounced "you-art") is a serial I/O peripheral that communicates between two systems without sending a clock. Instead, the systems must agree in advance about what data rate to use and must each locally generate its own clock. Hence, the transmission is asynchronous because the clocks are not synchronized. Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication. UARTs are used in protocols such as RS-232 and RS-485. For example, old computer serial ports use the RS-232C standard, introduced in 1969 by the Electronics Industries Associations. The standard originally envisioned connecting *Data Terminal Equipment* (DTE) such as a mainframe computer to *Data Communication Equipment* (DCE) such as a modem. Although a UART is relatively slow compared to SPI and prone to misconfiguration issues, the standards have been around for so long that they remain important today.

Figure e9.11(a) shows an asynchronous serial link. The DTE sends data to the DCE over the TX line and receives data back over the RX line. Figure e9.11(b) shows one of these lines sending a character at a data rate of 9600 baud. The lines idle at a logic '1' when not in use. Each character is sent as a start bit (0), 7 or 8 data bits, an optional parity bit, and one or more stop bits (1's). The UART detects the falling transition from idle to start to lock on to the transmission at the appropriate time. Although seven data bits is sufficient to send an ASCII character, eight bits are normally used because they can convey an arbitrary byte of data.

The optional parity bit allows the system to detect if a bit was corrupted during transmission. It can be configured as *even* or *odd*; even parity means that the parity bit is chosen such that the total collection of data and parity has an even number of 1's; in other words, the parity bit is the XOR of the data bits. The receiver can then check if an even number of 1's was received and signal an error if not. Odd parity is the reverse.

Baud rate gives the signaling rate, measured in symbols per second, whereas bit rate gives the data rate, measured in bits per second. The signaling we've discussed in this text is 2-level signaling, where each symbol represents a bit. However, multi-level signaling can send multiple bits per symbol; for example, 4-level signaling sends two bits per symbol. In that case, the bit rate is twice the baud rate. In a simple system like SPI where each symbol is a bit and each symbol represents data, the baud rate is equal to the bit rate. UARTs and some other signaling conventions require overhead bits in addition to the data. For example, a two-level signaling system that adds start and stop bits for each 8 bits of data and operates at a baud rate of 9600 has a bit rate of (9600 symbols/second)×(8 bits/10 symbols) = 7680 bits/second = 960 characters/second.

In the 1950s through 1970s, early hackers calling themselves phone phreaks learned to control the phone company switches by whistling appropriate tones. A 2600 Hz tone produced by a toy whistle from a Cap'n Crunch cereal box e9.12 could be exploited to place free long-distance and international calls.

CAP'N
CRUNCH
BO'SUN WHISTLE

Figure e9.12 Cap'n Crunch Bosun Whistle
(Photograph by Evrim Sen, reprinted with permission.)

Handshaking refers to the negotiation between two systems; typically, one system signals that it is ready to send or receive data, and the other system acknowledges that request.

A common choice is 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information. Hence, signaling rates are referred to in units of baud rather than bits/sec. For example, 9600 baud indicates 9600 symbols/sec, or 960 characters/sec,. Both systems must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled. This is a hassle, especially for nontechnical users, which is one of the reasons that the Universal Serial Bus (USB) has replaced UARTs in personal computer systems.

Typical baud rates include 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200. The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones. In contemporary systems, 9600 and 115200 are two of the most common baud rates; 9600 is encountered where speed doesn't matter, and 115200 is the fastest standard rate, though still slow compared to other modern serial I/O standards.

The RS-232 standard defines several additional signals. The Request to Send (RTS) and Clear to Send (CTS) signals can be used for *hardware handshaking*. They can be operated in either of two modes. In *flow control* mode, the DTE clears RTS to 0 when it is ready to accept data from the DCE. Likewise, the DCE clears CTS to 0 when it is ready to receive data from the DTE. Some datasheets use an overbar to indicate that they are active-low. In the older *simplex* mode, the DTE clears RTS to 0 when it is ready to transmit. The DCE replies by clearing CTS when it is ready to receive the transmission.

Some systems, especially those connected over a telephone line, also use Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR), and Ring Indicator (RI) to indicate when equipment is connected to the line.

The original standard recommended a massive 25-pin DB-25 connector, but PCs streamlined to a male 9-pin DE-9 connector with the pinout shown in Figure e9.13(a). The cable wires normally connect straight across as shown in Figure e9.13(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure e9.13(c) may be needed to swap RX and TX and complete the handshaking. As a final insult, some connectors are male and some are female. In summary, it can take a large box of cables and a certain amount of guess-work to connect two systems over RS-232, again explaining the shift to USB. Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS.

RS-232 represents a 0 electrically with 3 to 15 V and a 1 with −3 to −15 V; this is called *bipolar* signaling. A transceiver converts the digital logic levels of the UART to the positive and negative levels expected by RS-232, and also provides electrostatic discharge protection to protect the serial port from getting zapped when the user plugs in a cable. The

MAX3232E is a popular transceiver compatible with both 3.3 and 5 V digital logic. It contains a charge pump that, in conjunction with external capacitors, generates ±5 V outputs from a single low-voltage power supply. Some serial peripherals intended for embedded systems omit the transceiver and just use 0 V for a 0 and 3.3 or 5 V for a 1; check the datasheet!

The BCM2835 has two UARTs named UART0 and UART1. Either can be configured to communicate on pins 14 and 15, but UART0 is more fully featured and is described here. To use these pins for UART0 rather than GPIO, their GPFSEL must be set to ALT0. As with SPI, the Pi must first configure the port. Unlike SPI, reading and writing can occur independently because either system may transmit without receiving and vice versa. UART0's registers are shown in Figure e9.14.

To configure the UART, first set the baud rate. The UART has an internal 3 MHz clock that must be divided down to produce a clock that is 16x the desired baud rate. Hence, the appropriate divisor, BRD, is

$$\mathrm{BRD} = 3000000/(16 \times \text{baud rate})$$

BRD is represented with a 16-bit integer portion in UART_IBRD and a 6-bit fractional portion in UART_FBRD: BRD = IBRD + FBRD/64. Table e9.6 shows these settings for popular baud rates.[1]

Table e9.6 BRD settings

Target Baud Rate	UART_IBRD	UART_FBRD	Actual Baud Rate	Error (%)
300	625	0	300	0
1200	156	16	1200	0
2400	78	8	2400	0
9600	19	34	9600	0
19200	9	49	19200	0
38400	4	56	38461	0.16
57600	3	16	57692	0.16
115200	1	40	115384	0.16

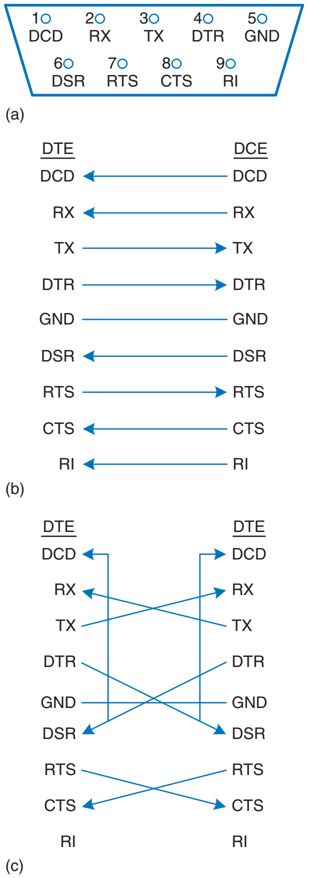

1 DCD
2 RX
3 TX
4 DTR
5 GND
6 DSR
7 RTS
8 CTS
9 RI
(a)
DTE
DCE
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
(b)
DTE
DTE
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
(c)

Figure e9.13 DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring

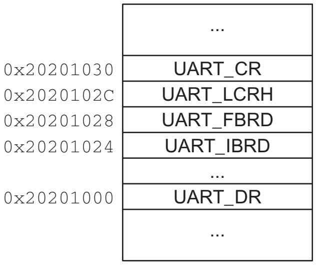

...
0x20201030 UART_CR
0x2020102C UART_LCRH
0x20201028 UART_FBRD
0x20201024 UART_IBRD
...
0x20201000 UART_DR
...

Figure e9.14 UART0 registers

---

[1] The baud rates do not all evenly divide 3 MHz, so some divisors produce a frequency error. The UART, by its asynchronous nature, accommodates this error so long as it is small enough.

Next, set the number of data, stop, and parity bits using the UART_LCRH line control register. By default, the UART has 1 stop bit and no parity, but strangely only transmits and receives 5-bit words. Hence, the WLEN field (bits 6:5) of UART_LCRH must be set to 3 to handle 8-bit words. Finally, enable the UART by turning on bit 0 (UARTEN) of the UART_CR control register.

Data is transmitted and received using the UART_DR data register and UART_FR framing register. To transmit data, wait until bit 7 (TXFE) of UART_FR is 1 to indicate that the transmitter is not busy, then write a byte to UART_DR. To receive data, wait until bit 4 (RXFE) of UART_FR is 0 to indicate that the receiver has data, then read the byte from UART_DR.

Example e9.5 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for a Raspberry Pi to communicate with a PC over a serial port at 115200 baud with 8 data bits, 1 stop bit, and no parity. The PC should be running a console program such as PuTTY[2] to read and write over the serial port. The program should ask the user to type a string. It should then tell the user what she typed.

Solution: Figure e9.15(a) shows a basic schematic of the serial link illustrating the issues of level conversion and cabling. Because few PCs still have physical serial ports, we use a Plugable USB to RS-232 DB9 Serial Adapter from `plugable.com` shown in Figure e9.16 to provide a serial connection to the PC. The adapter connects to a female DE-9 connector soldered to wires that feed a transceiver, which converts the voltages from the bipolar RS-232 levels to the Pi's 3.3 V level. The Pi and PC are both Data Terminal Equipment, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the Pi is not used, and the RTS and CTS on the DE9 connector are tied together so that the PC will shake its own hand. Figure e9.15(b) shows an easier approach with an Adafruit 954 USB to TTL serial cable. The cable is directly compatible with 3.3 V levels and has female header pins that plug directly into the Raspberry Pi male headers.

To configure PuTTY to work with the serial link, set *Connection type* to Serial and *Speed* to 115200. Set *Serial line* to the COM port assigned by the operating system to the Serial to USB Adapter. In Windows, this can be found

---

[2] PuTTY is available for free download at `www.putty.org`.

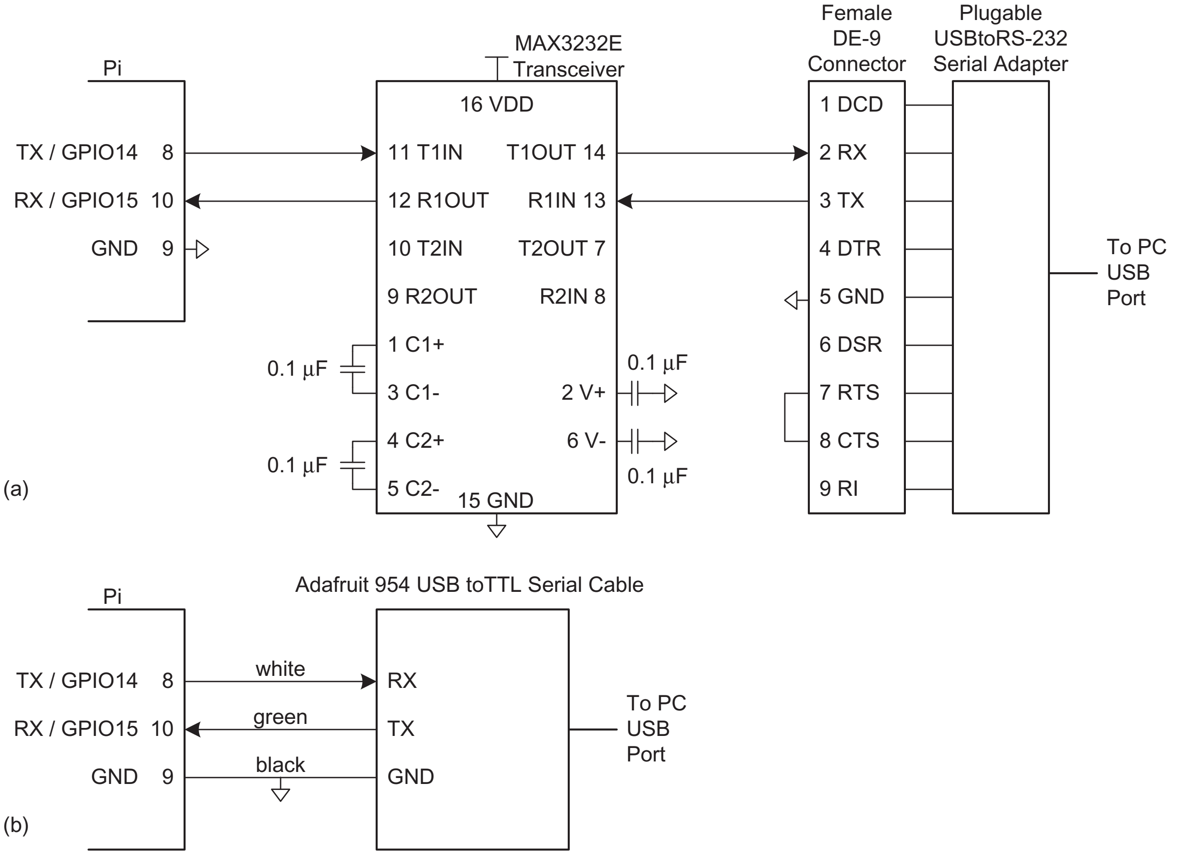

Pi
TX / GPIO14 8
RX / GPIO15 10
GND 9
MAX3232E Transceiver
16 VDD
11 T1IN
T1OUT 14
12 R1OUT
R1IN 13
10 T2IN
T2OUT 7
9 R2OUT
R2IN 8
1 C1+
3 C1-
2 V+
4 C2+
6 V-
5 C2-
15 GND
0.1 µF
0.1 µF
0.1 µF
0.1 µF
Female DE-9 Connector
1 DCD
2 RX
3 TX
4 DTR
5 GND
6 DSR
7 RTS
8 CTS
9 RI
Plugable USBtoRS-232 Serial Adapter
To PC USB Port
(a)
Pi
Adafruit 954 USB toTTL Serial Cable
TX / GPIO14 8
RX / GPIO15 10
GND 9
white
green
black
RX
TX
GND
To PC USB Port
(b)

Figure e9.15 **Raspberry Pi to PC serial link (a) Plugable cable, (b) Adafruit cable**

in the Device Manager; for example, it might be COM3. Under the *Connection Serial* tab, set flow control to NONE or RTS/CTS. Under the Terminal tab, set Local Echo to Force On to have characters appear in the terminal as you type them.

The serial port device driver code in EasyPIO.h is listed below. The Enter key in the terminal program corresponds to a carriage return character represented as `'\r'` in C with an ASCII code of 0x0D. To advance to the beginning of the next line when printing, send both the `'\n'` and `'\r'` (new line and carriage return) characters.[3] The `uartInit` function configures the UART as described above. Similarly, `getCharSerial` and `putCharSerial`

Note that the operating system also prints a log-in prompt to the serial port. You may see some interesting interactions between the OS and your program when both use the port.

Figure e9.16 **Plugable USB to RS-232 DB9 serial adapter**

(© 2012 Plugable Technologies; reprinted with permission.)

wait until the UART is ready and then read or write a byte from the data register.

```
void uartInit(int baud) {
    uint fb = 12000000/baud;        // 3 MHz UART clock

    pinMode(14, ALT0);              // TX
    pinMode(15, ALT0);              // RX
    UART_IBRD = fb >> 6;            // 6 Fract, 16 Int bits of BRD
    UART_FBRD = fb & 63;
    UART_LCRHbits.WLEN = 3;         // 8 Data, 1 Stop, no Parity, no FIFO, no Flow
    UART_CRbits.UARTEN = 1;         // Enable uart
}

char getCharSerial(void) {
    while (UART_FRbits.RXFE);       // Wait until data is available
    return UART_DRbits.DATA;        // Return char from serial port
}

void putCharSerial(char c) {
    while (!UART_FRbits.TXFE);      // Wait until ready to transmit
    UART_DRbits.DATA = c;           // Send char to serial port
}
```

The `main` function demonstrates printing to the console and reading from the console using the `putStrSerial` and `getStrSerial` functions.

```
#include "EasyPIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
   int i = 0;
   do {                                              // Read an entire string until
      str[i] = getCharSerial();                      // Carriage return
   } while ((str[i++] != '\r') && (i < MAX_STR_LEN)); // Look for carriage return
   str[i-1] = 0;                                     // Null-terminate the string
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {                           // Iterate over string
       putCharSerial(str[i++]);                     // Send each character
    }
}
```

```
int main(void) {
    char str[MAX_STR_LEN];
```

---

[3] PuTTY prints correctly even if the `\r` is omitted.

```
    pioInit();
    uartInit(115200);           // Initialize UART with baud rate

    while (1) {
        putStrSerial("Please type something: \r\n");
        getStrSerial(str);
        putStrSerial("You typed: ");
        putStrSerial(str);
        putStrSerial("\r\n");
    }
}
```

Communicating with the serial port from a C program on a PC is a bit of a hassle because serial port driver libraries are not standardized across operating systems. Other programming environments such as Python, Matlab, or LabVIEW make serial communication painless.

9.3.5 Timers

Embedded systems commonly need to measure time. For example, a microwave oven needs a timer to keep track of the time of day and another to measure how long to cook. It might use yet another to generate pulses to the motor spinning the platter, and a fourth to control the power setting by only activating the microwave's energy for a fraction of every second.

The BCM2835 has a system timer with a 64-bit free-running counter that increments every microsecond (i.e. at 1 MHz) and four 32-bit timer compare channels. Figure e9.17 shows the memory map for the system timer. SYS_TIMER_CLO and CHI contain the lower and upper 32 bits of the 64-bit counter value. SYS_TIMER_C0…C3 are 32-bit compare channels. When any of the compare channels match SYS_TIMER_CLO, the corresponding match bit (M0-M3) in the bottom four bits of SYS_TIMER_CS is set. A match bit is cleared by writing a 1 to that bit of SYS_TIMER_CS. This may seem counterintuitive, but it prevents inadvertently clearing other match bits. One can measure a particular number of microseconds by adding that time to CLO and putting it in C1, clearing SYS_TIMER_CS.M1, then waiting until SYS_TIMER_CS.M1 is set.

The graphics processing unit and operating system may use channels 0, 2, and 3, so user code should check SYSTEM_TIMER_C1.

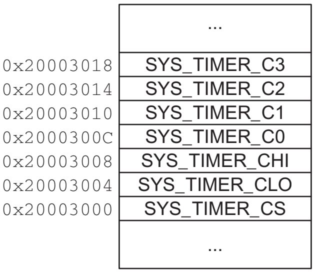


Figure e9.17 System timer registers

Unfortunately, Linux is a multitasking operating system that may switch between processes without warning. If your program is waiting for a timer match and then another process begins executing, your program may not resume until long after the match occurs and you may measure the wrong amount of time. To avoid this, your program can turn off interrupts during critical timing loops so that Linux will not switch processes. Be sure to turn the interrupts back on when you are done. EasyPIO defines `noInterrupts` and `interrupts` functions to disable and enable interrupts, respectively. While interrupts are disabled, the Pi will not switch between processes and cannot even respond to the user pressing Ctrl-C to kill a program. If your program hangs, you'll need to turn off power and reboot your Pi to recover.

Example e9.6 BLINKING LED

Write a program that blinks the status LED on the Raspberry Pi 5 times per second for 4 seconds.

Solution: The `delayMicros` function in EasyPIO creates a delay of a specified number of microseconds using the timer compare channel 1.

```
void delayMicros(int micros) {
    SYS_TIMER_C1 = SYS_TIMER_CLO + micros;  // Set the compare register
    SYS_TIMER_CSbits.M1 = 1;                // Reset match flag to 0
    while (SYS_TIMER_CSbits.M1 == 0);       // Wait until match flag is set
}

void delayMillis(int millis) {
    delayMicros(millis*1000);               // 1000 μs per ms
}
```

GPIO47 drives the activity LED on the Pi B+. The program sets this pin to be an output and disables interrupts. It then turns the LED OFF and ON through a series of digital writes with a 200 ms repetition rate (5 Hz). The program finally reenables interrupts.

```
#include "EasyPIO.h"

void main(void) {
   int i;

   pioInit();

   pinMode(47, OUTPUT);      // Status led as output
   noInterrupts();           // Disable interrupts

   for (i=0; i<20; i++) {
       delayMillis(150);
       digitalWrite(47, 0); // Turn led off
       delayMillis(50);
       digitalWrite(47, 1); // Turn led on
   }
   interrupts();             // Re-enable interrupts
}
```

9.3.6 Analog I/O

The real world is an analog place. Many embedded systems need analog inputs and outputs to interface with the world. They use analog-to-digital converters (ADCs) to quantize analog signals into digital values, and digital-to-analog-converters (DACs) to do the reverse. Figure e9.18 shows symbols for these components. Such converters are characterized by their resolution, dynamic range, sampling rate, and accuracy. For example, an ADC might have $N=12$-bit resolution over a range V_{ref}^{-} to V_{ref}^{+} of 0–5 V with a sampling rate of $f_s = 44$ kHz and an accuracy of ±3 least significant bits (lsbs). Sampling rates are also listed in samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{in}(t)$ and the digital sample $X[n = t / f_s]$ is

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}}$$

For example, an input voltage of 2.5 V (half of full scale) would correspond to an output of $100000000000_2 = 800_{16}$, with an uncertainty of up to 3 lsbs.

Similarly, a DAC might have $N=16$-bit resolution over a full-scale output range of $V_{ref} = 2.56$ V. It produces an output of

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref}$$

Many microcontrollers have built-in ADCs of moderate performance. For higher performance (e.g., 16-bit resolution or sampling rates in excess of 1 MHz), it is often necessary to use a separate ADC connected to the microcontroller. Fewer microcontrollers have built-in DACs, so separate chips may also be used. However, microcontrollers often produce analog outputs using a technique called pulse-width modulation (PWM).

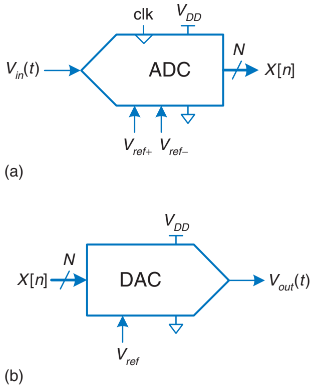


Figure e9.18 ADC and DAC symbols

9.3.6.1 D/A Conversion

The BCM2835 has a specialized DAC for composite video output, but no general-purpose converter. This section describes D/A conversion using external DACs and illustrates interfacing the Raspberry Pi to other chips over parallel and serial ports. The next section achieves the same result using pulse-width modulation.

Some DACs accept the N-bit digital input on a parallel interface with N wires, while others accept it over a serial interface such as SPI. Some DACs require both positive and negative power supply voltages, while others operate off of a single supply. Some support a flexible range of supply voltages, while others demand a specific voltage. The input logic levels should be compatible with the digital source. Some DACs produce a voltage output proportional to the digital input, while others produce a current output; an operational amplifier may be needed to convert this current to a voltage in the desired range.

In this section, we use the Analog Devices AD558 8-bit parallel DAC and the Linear Technology LTC1257 12-bit serial DAC. Both produce voltage outputs, run off a single 5–15 V power supply, use $V_{IH} = 2.4$ V such that they are compatible with 3.3 V I/O, come in DIP packages that make them easy to breadboard, and are easy to use. The AD558 produces an output on a scale of 0-2.56 V, consumes 75 mW, comes in a 16-pin package, and has a 1 μs settling time permitting an output rate of 1 Msamples/sec. The datasheet is at analog.com. The LTC1257 produces an output on a scale of 0–2.048 V, consumes less than 2 mW, comes in an 8-pin package, and has a 6 μs settling time. Its SPI operates at a maximum of 1.4 MHz. The datasheet is at linear.com.

Example e9.7 ANALOG OUTPUT WITH EXTERNAL DACS

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a Raspberry Pi, an AD558, and an LTC1257.

Solution: The circuit is shown in Figure e9.19. The AD558 connects to the Pi via GPIO14, 15, 17, 18, 22, 23, 24, and 25. It connects *Vout Sense* and *Vout Select* to *Vout* to set the 2.56 V full-scale output range. The LTC1257 connects

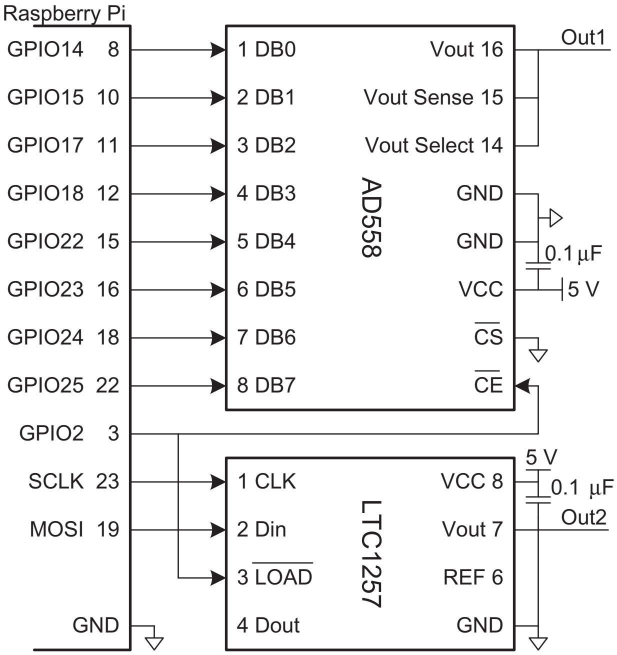

Raspberry Pi
GPIO14 8
GPIO15 10
GPIO17 11
GPIO18 12
GPIO22 15
GPIO23 16
GPIO24 18
GPIO25 22
GPIO2 3
SCLK 23
MOSI 19
GND
1 DB0
2 DB1
3 DB2
4 DB3
5 DB4
6 DB5
7 DB6
8 DB7
AD558
Vout 16
Vout Sense 15
Vout Select 14
GND
GND
VCC
CS
CE
Out1
0.1 μF
5 V
1 CLK
2 Din
3 LOAD
4 Dout
LTC1257
VCC 8
Vout 7
REF 6
GND
5 V
0.1 μF
Out2

Figure e9.19 **DAC parallel and serial interfaces to a Raspberry Pi**

to the Pi via SPI0. Both ADCs use a 5 V power supply and have a 0.1 μF decoupling capacitor to reduce power supply noise. The active-low chip enable and load signals on the DACs indicate when to convert the next digital input. They are driven high while a new input is being loaded.

The program is listed below. `pinsMode` and `digitalWrites` are similar to `pinMode` and `digitalWrite` but operate on an array of pins. The program sets the 8 parallel port pins to be outputs and also configures GPIO2 as an output to drive the chip enable and load signals. It initializes the SPI to 1.4 MHz. `initWaveTables` precomputes an array of sample values for the sine and triangle waves. The sine wave is set to a 12-bit scale and the triangle to an 8-bit scale. There are 64 points per period of each wave; changing this value trades precision for frequency. `genWaves` cycles through the samples. It disables interrupts to avoid switching processes and garbling the waves. For each sample, it disables the CE and LOAD signals to the DACs, sends the new sample over the parallel and serial ports, reenables the DACs, and then waits until the timer indicates that it is time for the next sample. `spiSendReceive16` transmits two bytes, but the LTC1257 only cares about the last 12 bits sent. The maximum frequency of somewhat more than 1000 Hz (64 Ksamples/sec) is set by the time to send each point in the `genWaves` function, of which the SPI transmission is a major component.

```
#include "EasyPIO.h"
#include math.h> // required to use the sine function

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];
int parallelPins[8] = {14,15,17,18,22,23,24,25};

void initWaveTables(void) {
    int i;
    for (i=0; i<NUMPTS; i++) {
        sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1);  // 12-bit scale
        if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;    //  8-bit scale
        else triangle[i] = 510-i*511/NUMPTS;
    }
}

void genWaves(int freq) {
    int i, j;
    int microPeriod = 1000000/(NUMPTS*freq);

    noInterrupts();     // disable interrupts to get accurate timing
    for (i=0; i<2000; i++){
        for (j=0; j<NUMPTS; j++) {
            SYS_TIMER_C1 = SYS_TIMER_CLO + microPeriod;  // Set time between samples
            SYS_TIMER_CSbits.M1 = 1;                     // Clear timer match
            digitalWrite(2,1);                           // No load while changing inputs
            spiSendReceive16(sine[j]);
            digitalWrites(parallelPins, 8, triangle[j]);
            digitalWrite(2,0);                           // Load new points into DACs
            while (!SYS_TIMER_CSbits.M1);                // Wait until timer matches
        }
    }
    interrupts();
}

void main(void) {
    pioInit();

    pinsMode(parallelPins, 8, OUTPUT);  // Set pins connected to the AD558 as outputs
    pinMode(2, OUTPUT);                 // Make pin 2 an output to control LOAD and CE
    spiInit(1400000, 0);                // 1.4MHz SPI clock, default settings
    initWaveTables();
    genWaves(1000);
}
```

9.3.6.2 Pulse-Width Modulation

Another way for a digital system to generate an analog output is with *pulse-width modulation* (PWM), in which a periodic output is pulsed high for part of the period and low for the remainder. The duty cycle is the fraction of the period for which the pulse is high, as shown in Figure e9.20. The average value of the output is proportional to the duty cycle. For example, if the output swings between 0 and 3.3 V and has a duty cycle of 25%, the average value will be $0.25 \times 3.3 = 0.825$ V. Low-pass filtering a PWM signal eliminates the oscillation and leaves a signal with the desired average value. Thus, PWM is an effective way to produce an analog output if the pulse rate is much higher than the analog output frequencies of interest.

The BCM2835 has a PWM controller capable of producing two simultaneous outputs. PWM0 is available at GPIO18 as pin function ALT5, while both PWM outputs are available on the stereo audio jack. Figure e9.21 shows the memory map for the PWM unit and for the clock manager that it depends upon.

The PWM_CTL register is used to turn on pulse width modulation. Bit 0 (PWEN1) must be set to enable the output. Bit 7 (MSEN1: mark-space enable) should also be set to produce pulse width modulation of the form of Figure e9.20 in which the output is HIGH for part of the period and LOW for the remainder.

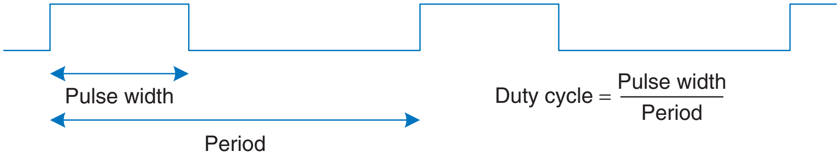


Figure e9.20 **Pulse-width modulated (PWM) signal**

The PWM signals are derived from a PWM clock generated by the BCM2835 clock manager. The PWM_RNG1 and PWM_DAT1 registers control the period and duty cycle, respectively, by specifying the number of PWM clock ticks for the overall waveform and for the HIGH portion. For example, if the clock manager produces a 25 MHz clock and PWM_RNG1 = 1000 and PWM_DAT1 = 300, the PWM output will operate at (25 MHz / 1000) = 25 kHz and the duty cycle will be 300 / 1000 = 30%.

The clock manager is configured using the CM_PWMCTL and the frequency is set using the CM_PWMDIV register. Table e9.7 summarizes the bit fields of the CM_PWMCTL register. The maximum frequency of the PWM clock is 25 MHz. It can be obtained from the 500 MHz PLLD clock on the Pi as follows:

- CM_PWMCTL: Write 0x5A to PASSWD and 1 to KILL to stop the clock generator
- CM_PWMCLT: Wait for BUSY to clear to indicate the clock is stopped
- CM_PWMCTL: Write 0x5A to PASSWD, 1 to MASH, and 6 to SRC to select PLLD with no audio noise shaping
- CM_PWMDIV: Write 0x5A to PASSWD and 20 to bits 23:12 to divide PLLD by 20 from 500 MHz down to 25 MHz
- CM_PWMCTL: Write 0x5A to PASSWD and 1 to ENAB to restart the clock generator
- CM_PWMCTL: Wait for BUSY to set to indicate the clock is running

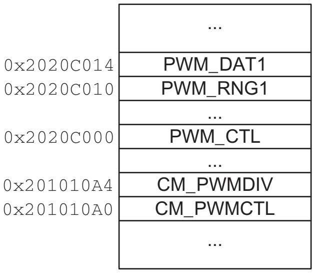


Figure e9.21 **PWM and clock manager registers**

The CM_PWM registers are not documented in the BCM2835 datasheet. You may find information on them by searching the Internet for "BCM2835 Audio & PWM Clocks" by G.J. van Loo.

Table e9.7 **CM_PWMCTL register fields**

Bit	Name	Description
31:24	PASSWD	Must be set to 5A when writing
10:9	MASH	Audio noise shaping
7	BUSY	Clock generator running
5	KILL	Write a 1 to stop the clock generator
4	ENAB	Write a 1 to start the clock generator
3:0	SRC	Clock source

Example e9.8 ANALOG OUTPUT WITH PWM

Write an `analogWrite(val)` function to generate an analog output voltage using PWM and an external RC filter. The function should accept an input between 0 (for 0 V output) and 255 (for full 3.3 V output).

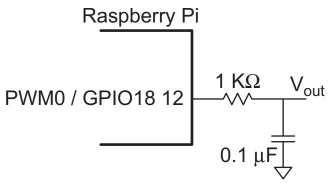


Figure e9.22 **Analog output using PWM and low-pass filter**

Solution: Use PWM0 to produce a 78.125 kHz signal on GPIO18. The low pass filter in Figure e9.22 has a corner frequency of

$$f_c = \frac{1}{2\pi RC} = 1.6 \text{ kHz}$$

to eliminate the high-speed oscillations and pass the average value.

The PWM functions in EasyPIO are given below. `pwmInit` initializes the PWM module on GPIO18 as described above. `setPWM` sets the frequency and duty cycle of the PWM output. Duty should be between 0 (always OFF) and 1 (always ON). The `analogWrite` function sets the duty cycle based on a full scale of 255.

```
    // Default PLLD value is 500 [MHz]
    #define PLL_FREQUENCY 500000000
    // Max pwm clk is 25 [MHz]
    #define CM_FREQUENCY 25000000
    #define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

void pwmInit() {
    pinMode(18, ALT5);

    // Configure the clock manager to generate a 25 MHz PWM clock.
    // Documentation on the clock manager is missing in the datasheet
    // but found in "BCM2835 Audio and PWM Clocks" by G.J. van Loo 6 Feb 2013.
    // Maximum operating frequency of PWM clock is 25 MHz.
    // Writes to the clock manager registers require simultaneous writing
    // a "password" of 5A to the top bits to reduce the risk of accidental writes.

    CM_PWMCTL = 0; // Turn off PWM before changing
    CM_PWMCTL = PWM_CLK_PASSWORD|0x20; // Turn off clock generator
    while (CM_PWMCTLbits.BUSY); // Wait for generator to stop
    CM_PWMCTL = PWM_CLK_PASSWORD|0x206; // Src = unfiltered 500 MHz CLKD
    CM_PWMDIV = PWM_CLK_PASSWORD|(PLL_CLOCK_DIVISOR << 12); // 25 MHz
    CM_PWMCTL = CM_PWMCTL|PWM_CLK_PASSWORD|0x10;  // Enable PWM clock
    while (!CM_PWMCTLbits.BUSY);  // Wait for generator to start
    PWM_CTLbits.MSEN1 = 1;  // Channel 1 in mark/space mode
    PWM_CTLbits.PWEN1 = 1;  // Enable PWM
}

void setPWM(float freq, float duty) {
    PWM_RNG1 = (int)(CM_FREQUENCY / freq);
    PWM_DAT1 = (int)(duty * (CM_FREQUENCY / freq));
}
```

```
void analogWrite(int val) {
  setPWM(78125, val/255.0);
}
```

The main function tests the PWM by setting the output to half scale (1.65 V).

```
#include "EasyPIO.h"

void main(void) {
  pioInit();
  pwmInit();
  analogWrite(128);
}
```

9.3.6.3 A/D Conversion

The BCM2835 has no built-in ADC, so this section describes A/D conversion using an external converter similar to the external DAC.

Example e9.9 ANALOG INPUT WITH AN EXTERNAL ADC

Interface a 10-bit MCP3002 A/D converter to a Raspberry Pi using SPI and print the input value. Set a full scale voltage of 3.3 V. Search for the datasheet on the Web for full details of operation.

Solution: Figure e9.23 shows a schematic of the connection. The MCP3002 uses VDD as its full scale reference. It accepts a 3.3–5.5 V supply and we

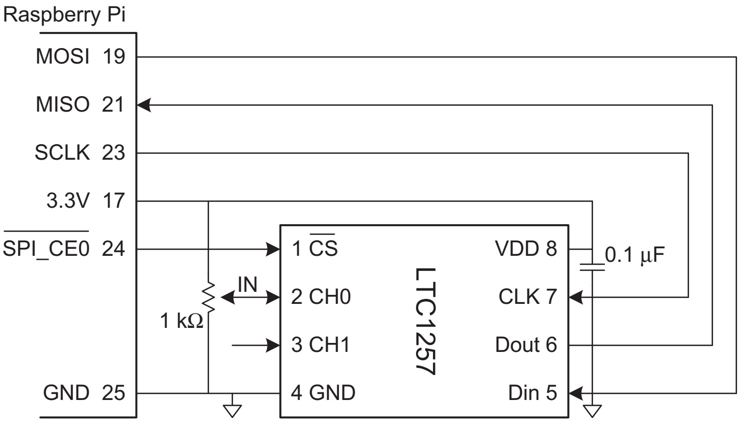

Raspberry Pi
MOSI 19
MISO 21
SCLK 23
3.3V 17
SPI_CE0 24
GND 25
1 kΩ
IN
1 CS
2 CH0
3 CH1
4 GND
LTC1257
VDD 8
CLK 7
Dout 6
Din 5
0.1 μF

Figure e9.23 Analog input using external ADC choose 3.3 V. The ADC has two input channels, and we connect channel 0 to a potentiometer that we can rotate to adjust the input voltage between 0 and 3.3 V.

The Pi code initializes the SPI and repeatedly reads and prints samples. According to the datasheet, the Raspberry Pi must send the 16-bit quantity 0x6000 over SPI to read CH0 and will receive the 10-bit result back in the bottom 10 bits of the 16-bit result. The converter also requires a chip select signal, conveniently provided by the SPI chip enable.

```
#include "EasyPIO.h"

void main(void) {
    int sample;

    pioInit();
    spiInit(200000, 0); // 200 kHz SPI clock, default settings

    while (1){
        sample = spiSendReceive16(0x6000);
        printf("Read %d\n", sample);
    }
}
```

9.3.7 Interrupts

So far, we have relied on *polling*, in which the program continually checks until an event occurs such as data arriving on a UART or a timer reaching its compare value. This can be a waste of the processor's power and makes it difficult to write programs that do interesting work while simultaneously waiting for events to occur.

Most microcontrollers support *interrupts*. When an event occurs, the microcontroller can stop regular program execution and jump to an interrupt handler that responds to the interrupt, then return seamlessly to where it left off.

The Raspberry Pi normally runs Linux, which intercepts interrupts before they get to the program. Therefore, it is presently not straightforward to write interrupt-based programs and this text does not provide examples on the Pi.

9.4 OTHER MICROCONTROLLER PERIPHERALS

Microcontrollers frequently interface with other external peripherals. This section describes a variety of common examples, including character-mode liquid crystal displays (LCDs), VGA monitors, Bluetooth wireless links, and motor control. Standard communication interfaces including USB and Ethernet are described in Sections 9.6.1 and 9.6.4.

9.4.1 Character LCDs

A character LCD is a small liquid crystal display capable of showing one or a few lines of text. They are commonly used in the front panels of appliances such as cash registers, laser printers, and fax machines that need to display a limited amount of information. They are easy to interface with a microcontroller over parallel, RS-232, or SPI interfaces. Crystalfontz America sells a wide variety of character LCDs ranging from 8 columns × 1 row to 40 columns × 4 rows with choices of color, backlight, 3.3 or 5 V operation, and daylight visibility. Their LCDs can cost $20 or more in small quantities, but prices come down to under $5 in high volume.

This section gives an example of interfacing a Raspberry Pi to a character LCD over an 8-bit parallel interface. The interface is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi. Figure e9.24 shows a Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD.

Figure e9.25 shows the LCD connected to a Pi over an 8-bit parallel interface. The logic operates at 5 V but is compatible with 3.3 V inputs from the Pi. The LCD contrast is set by a second voltage produced with a potentiometer; it is usually most readable at a setting of 4.2–4.8 V. The LCD receives three control signals: RS (1 for characters, 0 for instructions), R/$\overline{\text{W}}$ (1 to read from the display, 0 to write), and E (pulsed high for at least 250 ns to enable the LCD when the next byte is ready). When the instruction is read, bit 7 returns the busy flag, indicating 1 when busy and 0 when the LCD is ready to accept another instruction.

To initialize the LCD, the Pi must write a sequence of instructions to the LCD as given in Table e9.8. The instructions are written by holding RS = 0 and R/$\overline{\text{W}}$ = 0, putting the value on the eight data lines, and pulsing E. After each instruction, it must wait for at least a specified amount of time (or sometimes until the busy flag is clear).

*** CRYSTALFONTZ ***
*** CFAH2002ATMI ***

Figure e9.24 **Crystalfontz CFAH2002A-TMI 20×2 character LCD**
(© 2012 Crystalfontz America; reprinted with permission.)

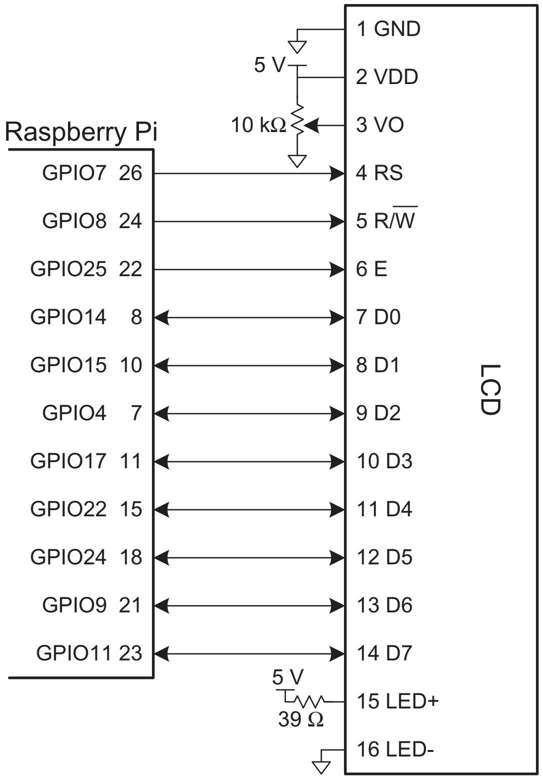

Raspberry Pi
GPIO7 26
GPIO8 24
GPIO25 22
GPIO14 8
GPIO15 10
GPIO4 7
GPIO17 11
GPIO22 15
GPIO24 18
GPIO9 21
GPIO11 23
5 V
10 kΩ
5 V
39 Ω
1 GND
2 VDD
3 VO
4 RS
5 R/$\overline{W}$
6 E
7 D0
8 D1
9 D2
10 D3
11 D4
12 D5
13 D6
14 D7
15 LED+
16 LED-
LCD

Figure e9.25 **Parallel LCD interface**

Table e9.8 **LCD initialization sequence**

Write	Purpose	Wait (µs)
(apply V_{DD})	Allow device to turn on	15000
0x30	Set 8-bit mode	4100
0x30	Set 8-bit mode again	100
0x30	Set 8-bit mode yet again	Until busy flag is clear
0x3C	Set 2 lines and 5 × 8 dot font	Until busy flag is clear
0x08	Turn display OFF	Until busy flag is clear
0x01	Clear display	1530
0x06	Set entry mode to increment cursor after each character	Until busy flag is clear
0x0C	Turn display ON with no cursor	

Then, to write text to the LCD, the Pi can send a sequence of ASCII characters. After each character, it must wait for the busy bit to clear. It may also send the instruction 0x01 to clear the display or 0x02 to return to the home position in the upper left.

Example e9.10 LCD CONTROL

Write a program to print "I love LCDs" to a character display.

Solution: The following program writes "I love LCDs" to the display by initializing the display and then sending the characters.

```
#include "EasyPIO.h"

int LCD_IO_Pins[] = {14, 15, 4, 17, 22, 24, 9, 11};

typedef enum {INSTR, DATA} mode;
#define RS 7
#define RW 8
#define E  25

char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md == DATA));      // Set instr/data mode
    digitalWrite(RW, 1);                // Read mode
    digitalWrite(E, 1);                 // Pulse enable
    delayMicros(10);                    // Wait for LCD response
    c = digitalReads(LCD_IO_Pins, 8);   // Read a byte from parallel port
    digitalWrite(E, 0);                 // Turn off enable
    delayMicros(10);
    return c;
}

void lcdBusyWait(void) {
    char state;
    do {
       state = lcdRead(INSTR);
    } while (state & 0x80);
}

void lcdWrite(char val, mode md) {
    pinsMode(LCD_IO_Pins, 8, OUTPUT);
    digitalWrite(RS, (md == DATA));     // Set instr/data mode. OUTPUT=1, INPUT=0
    digitalWrite(RW, 0);                // Set RW pin to write  (aka: 0)
    digitalWrites(LCD_IO_Pins, 8, val); // Write the char to the parallel port
    digitalWrite(E, 1); delayMicros(10); // Pulse E
    digitalWrite(E, 0); delayMicros(10);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delayMicros(1530);
}
```

```
void lcdPrintString(char* str) {
    while (*str != 0) {
        lcdWrite(*str, DATA); lcdBusyWait();
        str++;
    }
}

void lcdInit(void) {
    pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
    // send initialization routine:
    delayMicros(15000);
    lcdWrite(0x30, INSTR); delayMicros(4100);
    lcdWrite(0x30, INSTR); delayMicros(100);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

void main(void) {
    pioInit();
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

9.4.2 VGA Monitor

A more flexible display option is to drive a computer monitor. The Raspberry Pi has built-in support for HDMI and Composite video output. This section explains the low-level details of driving a VGA monitor directly from an FPGA.

The *Video Graphics Array* (VGA) monitor standard was introduced in 1987 for the IBM PS/2 computers, with a 640×480 pixel resolution on a *cathode ray tube* (CRT) and a 15-pin connector conveying color information with analog voltages. Modern LCD monitors have higher resolution but remain backward compatible with the VGA standard.

In a cathode ray tube, an electron gun scans across the screen from left to right exciting fluorescent material to display an image. Color CRTs use three different phosphors for red, green, and blue, and three electron beams. The strength of each beam determines the intensity of each color in the pixel. At the end of each scanline, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scanlines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. The process repeats about 60–75 times per second to refresh the fluorescence and give the visual illusion of a steady image. A liquid crystal display doesn't require the same electron scan gun, but uses the same VGA interface timing for compatibility.

In a 640×480 pixel VGA monitor refreshed at 59.94 Hz, the pixel clock operates at 25.175 MHz, so each pixel is 39.72 ns wide. The full screen can be viewed as 525 horizontal scanlines of 800 pixels each, but only 480 of the scanlines and 640 pixels per scan line actually convey the image, while the remainder are black. A scanline begins with a *back porch*, the blank section on the left edge of the screen. It then contains 640 pixels, followed by a blank *front porch* at the right edge of the screen and a horizontal sync (hsync) pulse to rapidly move the gun back to the left edge.

Figure e9.26(a) shows the timing of each of these portions of the scanline, beginning with the active pixels. The entire scan line is 31.778 μs long. In the vertical direction, the screen starts with a back porch at the top, followed by 480 active scan lines, followed by a front porch at the bottom and a vertical sync (vsync) pulse to return to the top to start the next frame. A new frame is drawn 60 times per second.

Figure e9.26(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. Higher resolutions use a faster pixel clock, up to 388 MHz at 2048×1536 at 85 Hz. For example, 1024×768 at 60 Hz can be achieved with a 65 MHz pixel clock.

The horizontal timing involves a front porch of 16 clocks, hsync pulse of 96 clocks, and back porch of 48 clocks. The vertical timing involves a front porch of 11 scan lines, vsync pulse of 2 lines, and back porch of 32 lines.

Figure e9.27 shows the pinout for a female connector coming from a video source. Pixel information is conveyed with three analog voltages for red, green, and blue. Each voltage ranges from 0–0.7 V, with more positive indicating brighter. The voltages should be 0 during the front and

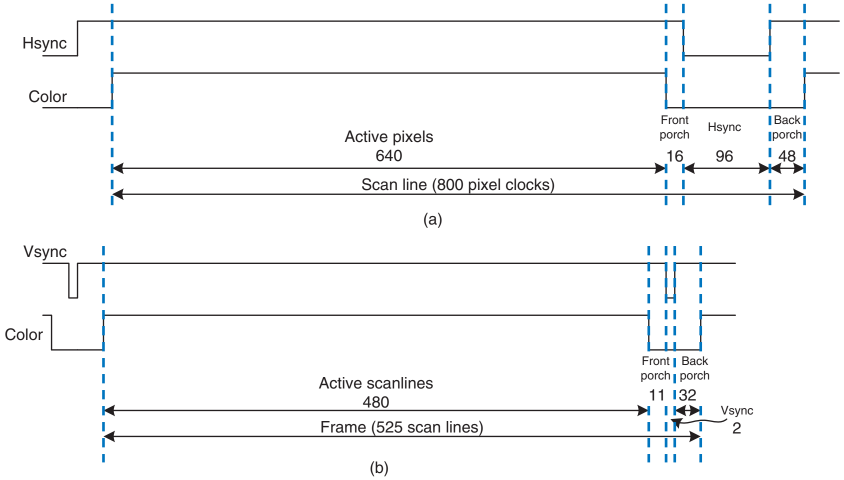

Hsync
Color
Active pixels
640
Front porch
16
Hsync
96
Back porch
48
Scan line (800 pixel clocks)
(a)
Vsync
Color
Active scanlines
480
Front porch
11
Back porch
32
Vsync
2
Frame (525 scan lines)
(b)

Figure e9.26 **VGA timing: (a) horizontal, (b) vertical**

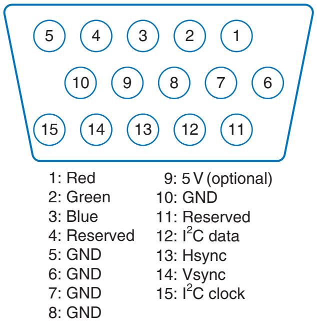

5 4 3 2 1
10 9 8 7 6
15 14 13 12 11
1: Red
2: Green
3: Blue
4: Reserved
5: GND
6: GND
7: GND
8: GND
9: 5 V (optional)
10: GND
11: Reserved
12: I^2C data
13: Hsync
14: Vsync
15: I^2C clock

Figure e9.27 **VGA connector pinout**

back porches. The video signal must be generated in real time at high speed, which is difficult on a microcontroller but easy on an FPGA. A simple black and white display could be produced by driving all three color pins with either 0 or 0.7 V using a voltage divider connected to a digital output pin. A color monitor, on the other hand, uses a *video DAC* with three separate D/A converters to independently drive the three color pins. Figure e9.28 shows an FPGA driving a VGA monitor through an ADV7125 triple 8-bit video DAC. The DAC receives 8 bits of R, G, and B from the FPGA. It also receives a SYNC_b signal that is driven active low whenever HSYNC or VSYNC are asserted. The video DAC produces three output currents to drive the red, green, and blue analog lines, which are normally 75 Ω transmission lines parallel terminated at both the video DAC and the monitor. The R_{SET} resistor sets the scale of the output current to achieve the full range of color. The clock rate depends on the resolution and refresh rate; it may be as high as 330 MHz with a fast-grade ADV7125JSTZ330 model DAC.

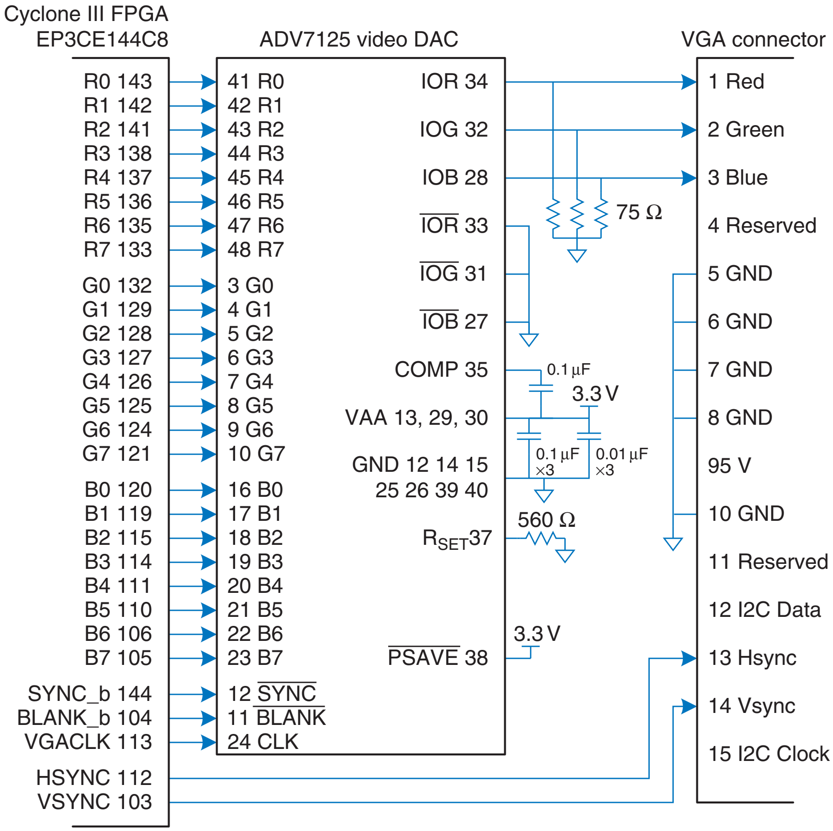

Cyclone III FPGA
EP3CE144C8
ADV7125 video DAC
VGA connector
R0 143
R1 142
R2 141
R3 138
R4 137
R5 136
R6 135
R7 133
G0 132
G1 129
G2 128
G3 127
G4 126
G5 125
G6 124
G7 121
B0 120
B1 119
B2 115
B3 114
B4 111
B5 110
B6 106
B7 105
SYNC_b 144
BLANK_b 104
VGACLK 113
HSYNC 112
VSYNC 103
41 R0
42 R1
43 R2
44 R3
45 R4
46 R5
47 R6
48 R7
3 G0
4 G1
5 G2
6 G3
7 G4
8 G5
9 G6
10 G7
16 B0
17 B1
18 B2
19 B3
20 B4
21 B5
22 B6
23 B7
12 SYNC
11 BLANK
24 CLK
IOR 34
IOG 32
IOB 28
IOR 33
IOG 31
IOB 27
COMP 35
VAA 13, 29, 30
GND 12 14 15 25 26 39 40
RSET37
PSAVE 38
75 Ω
0.1 μF
3.3 V
0.1 μF ×3
0.01 μF ×3
560 Ω
3.3 V
1 Red
2 Green
3 Blue
4 Reserved
5 GND
6 GND
7 GND
8 GND
95 V
10 GND
11 Reserved
12 I2C Data
13 Hsync
14 Vsync
15 I2C Clock

Figure e9.28 **FPGA driving VGA cable through video DAC**

Example e9.11 VGA MONITOR DISPLAY

Write HDL code to display text and a green box on a VGA monitor using the circuitry from Figure e9.28.

Solution: The code assumes a system clock frequency of 40 MHz and uses a phase-locked loop (PLL) on the FPGA to generate the 25.175 MHz VGA clock. PLL configuration varies among FPGAs; for the Cyclone III, the frequencies are specified with Altera's megafunction wizard. Alternatively, the VGA clock could be provided directly from a signal generator.

The VGA controller counts through the columns and rows of the screen, generating the hsync and vsync signals at the appropriate times. It also produces a blank_b signal that is asserted low to draw black when the coordinates are outside the 640×480 active region.

The video generator produces red, green, and blue color values based on the current (x, y) pixel location. (0, 0) represents the upper left corner. The generator draws a set of characters on the screen, along with a green rectangle. The character generator draws an 8×8-pixel character, giving a screen size of 80×60 characters. It looks up the character from a ROM, where it is encoded in binary as 6 columns by 8 rows. The other two columns are blank. The bit order is reversed by the SystemVerilog code because the leftmost column in the ROM file is the most significant bit, while it should be drawn in the least significant x-position.

Figure e9.29 shows a photograph of the VGA monitor while running this program. The rows of letters alternate red and blue. A green box overlays part of the image.

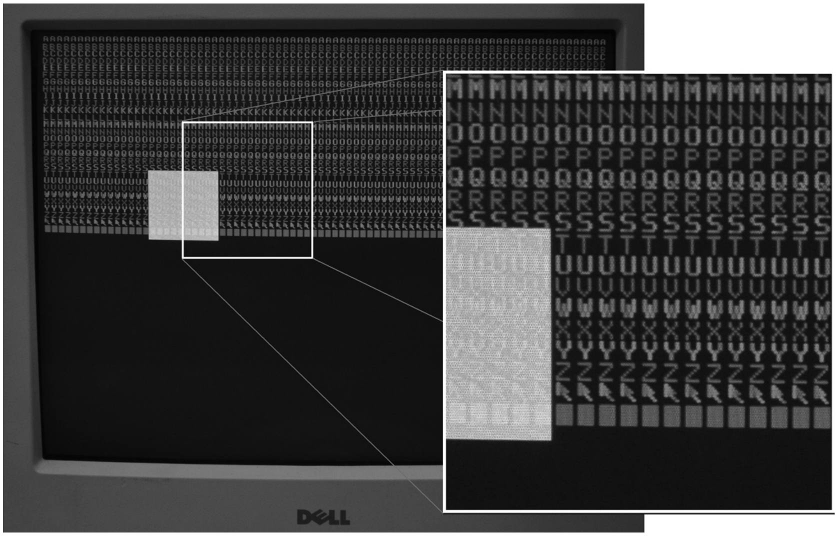


Figure e9.29 VGA output

vga.sv

```
module vga(input  logic       clk,
           output logic       vgaclk,          // 25.175 MHz VGA clock
           output logic       hsync, vsync,
           output logic       sync_b, blank_b, // To monitor & DAC
           output logic [7:0] r, g, b);        // To video DAC

  logic [9:0] x, y;

  // Use a PLL to create the 25.175 MHz VGA pixel clock
  // 25.175 MHz clk period = 39.772 ns
  // Screen is 800 clocks wide by 525 tall, but only 640 x 480 used
  // HSync = 1/(39.772 ns *800) = 31.470 kHz
  // Vsync = 31.474 kHz / 525 = 59.94 Hz (~60 Hz refresh rate)
  pll vgapll(.inclk0(clk), .c0(vgaclk));

  // Generate monitor timing signals
  vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

  // User-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE + VFP + VSYN + VBP)
                      (input  logic       vgaclk,
                       output logic       hsync, vsync, sync_b, blank_b,
                       output logic [9:0] x, y);
  // counters for horizontal and vertical positions
  always @(posedge vgaclk) begin
    x++;
    if (x == HMAX) begin
      x = 0;
      y++;
      if (y == VMAX) y = 0;
    end
  end

  // Compute sync signals (active low)
  assign hsync = ~(hcnt >= HACTIVE + HFP & hcnt < HACTIVE + HFP + HSYN);
  assign vsync = ~(vcnt >= VACTIVE + VFP & vcnt < VACTIVE + VFP + VSYN);
  assign sync_b = hsync & vsync;

  // Force outputs to black when outside the legal display area
  assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(inputlogic [9:0] x, y, output logic [7:0] r, g, b);
```

```
  logic     pixel, inrect;

  // Given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue.Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen  rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
  assign g =       inrect ? 8'hFF : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;            // a line read from the ROM

  // Initialize ROM with characters from text file
  initial
    $readmemb("charrom.txt", charrom);

  // Index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}]; // Subtract 65 because A
                                                // is entry 0

  // Reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic       inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
```

```
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

Bluetooth is named for King Harald Bluetooth of Denmark, a 10th century monarch who unified the warring Danish tribes. This wireless standard is only partially successful at unifying a host of competing wireless protocols!

9.4.3 Bluetooth Wireless Communication

There are many standards now available for wireless communication, including Wi-Fi, ZigBee, and Bluetooth. The standards are elaborate and require sophisticated integrated circuits, but a growing assortment of modules abstract away the complexity and give the user a simple interface for wireless communication. One of these modules is the BlueSMiRF, which is an easy-to-use Bluetooth wireless interface that can be used instead of a serial cable.

Bluetooth is a wireless standard initially developed by Ericsson in 1994 for low-power, moderate speed communication over distances of 5–100 meters, depending on the transmitter power level. It is commonly used to connect an earpiece to a cellphone or a keyboard to a computer. Unlike infrared communication links, it does not require a direct line of sight between devices.

Bluetooth operates in the 2.4 GHz unlicensed industrial-scientific-medical (ISM) band. It defines 79 radio channels spaced at 1 MHz intervals starting at 2402 MHz. It hops between these channels in a pseudo-random pattern to avoid consistent interference with other devices, such as wireless routers operating in the same band. As given in Table e9.9, Bluetooth transmitters are classified at one of three power levels, which dictate the range and power consumption. In the basic rate mode, it operates at 1 Mbit/sec using Gaussian frequency shift keying (FSK). In ordinary FSK, each bit is conveyed by transmitting a frequency of $f_c \pm f_d$, where f_c is the center frequency of the channel and f_d is an offset of at least 115 kHz. The abrupt transition in frequencies between bits consumes extra bandwidth. In Gaussian FSK, the change in frequency is smoothed to make better use of the spectrum. Figure e9.30 shows the frequencies being transmitted for a sequence of 0's and 1's on a 2402 MHz channel using FSK and GFSK.

A BlueSMiRF Silver module, shown in Figure e9.31(a), contains a Class 2 Bluetooth radio, modem, and interface circuitry on a small card with a serial interface. It communicates with another Bluetooth device

Table e9.9 Bluetooth classes

Class	Transmitter Power (mW)	Range (m)
1	100	100
2	2.5	10
3	1	5

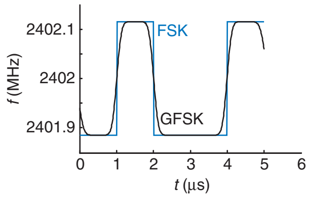


Figure e9.30 FSK and GFSK waveforms

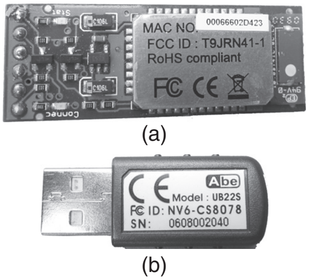


Figure e9.31 BlueSMiRF module and USB dongle

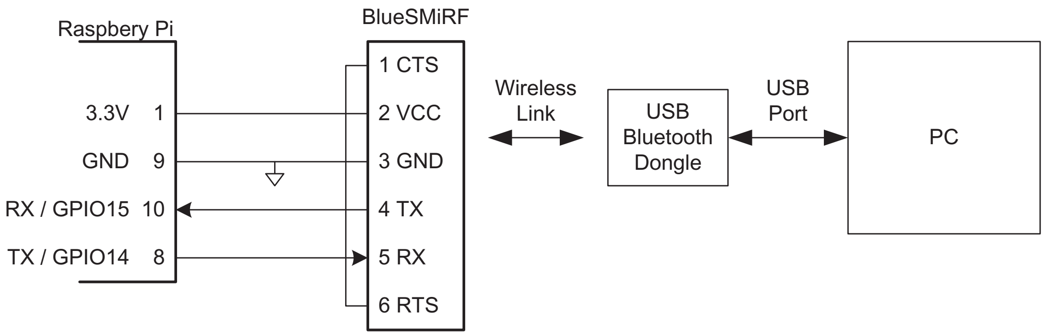


Figure e9.32 BlueSMiRF Raspberry Pi to PC link

such as a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a Pi and a PC similar to the link from Figure e9.15 but without the cable. The wireless link is compatible with the same software as is the wired link.

Figure e9.32 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the Pi, and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

The BlueSMiRF defaults to 115.2 k baud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

To use the interface, plug a USB Bluetooth dongle into a PC. Power up the Pi and BlueSMiRF. The red STAT light will flash on the BlueSMiRF indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with the BlueSMiRF. The default passkey for the BlueSMiRF is 1234. Take note of which COM port is assigned to the dongle. Then communication can proceed just as it would over a serial cable. Note that the dongle typically operates at 9600 baud and that PuTTY must be configured accordingly.

9.4.4 Motor Control

Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, so a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. They also require a *shaft encoder* if the user wants to know the current position of the motor. *Servo motors* accept a pulse-width modulated signal to specify their position over a limited range of angles. They are very easy to interface, but are not as powerful and are not suited to continuous rotation. *Stepper motors* accept a sequence of pulses, each of which rotates the motor by a fixed angle called a step. They are more expensive and still need an H-bridge to drive the high current, but the position can be precisely controlled.

Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.

9.4.4.1 DC Motors

Figure e9.33 shows the structure of a brushed DC motor. The motor is a two terminal device. It contains permanent stationary magnets called the *stator* and a rotating electromagnet called the *rotor* or *armature* connected to the shaft. The front end of the rotor connects to a split metal ring called a *commutator*. Metal brushes attached to the power lugs (input terminals) rub against the commutator, providing current to the rotor's electromagnet. This induces a magnetic field in the rotor that causes the rotor to spin to become aligned with the stator field. Once the rotor has spun part way around and approaches alignment with the stator, the brushes touch the opposite sides of the commutator, reversing the current flow and magnetic field and causing it to continue spinning indefinitely.

DC motors tend to spin at thousands of rotations per minute (RPM) at very low torque. Most systems add a gear train to reduce the speed to a more reasonable level and increase the torque. Look for a gear train designed to mate with your motor. Pittman manufactures a wide range of high quality DC motors and accessories, while inexpensive toy motors are popular among hobbyists.

A DC motor requires substantial current and voltage to deliver significant power to a load. The current should be reversible so the motor can spin in both directions. Most microcontrollers cannot produce enough current to drive a DC motor directly. Instead, they use an H-bridge, which conceptually contains four electrically controlled switches, as shown in Figure e9.34(a). If switches A and D are closed, current flows from left to right through the motor and it spins in one direction. If B and C are closed, current flows from right to left through the motor and it spins in the other direction. If A and C or B and D are closed, the voltage across the motor is forced to 0, causing the motor to actively brake. If none of the switches are closed, the motor will coast to a stop. The switches in an H-bridge are power transistors. The H-bridge also contains some digital logic to conveniently control the switches.

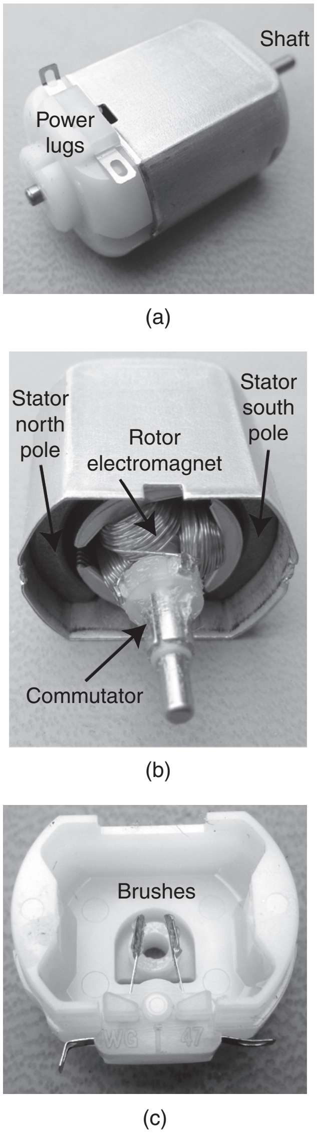


Figure e9.33 **DC motor**

When the motor current changes abruptly, the inductance of the motor's electromagnet will induce a large voltage spike that could damage the power transistors. Therefore, many H-bridges also have protection diodes in parallel with the switches, as shown in Figure e9.34(b). If the inductive kick drives either terminal of the motor above V_{motor} or below ground, the diodes will turn ON and clamp the voltage at a safe level. H-bridges can dissipate large amounts of power so a heat sink may be necessary to keep them cool.

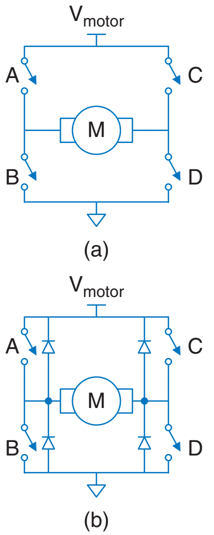

V_motor
A
C
M
B
D
(a)
V_motor
A
C
M
B
D
(b)

Figure e9.34 **H-bridge**

Example e9.12 AUTONOMOUS VEHICLE

Design a system in which a Raspberry Pi controls two drive motors for a robot car. Write a library of functions to initialize the motor driver and to make the car drive forward and back, turn left or right, and stop. Use PWM to control the speed of the motors.

Solution: Figure e9.35 shows a pair of DC motors controlled by a Pi via a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5–36 V motor supply V_{CC2}; it has $V_{IH} = 2$ V and is hence compatible with the 3.3 V I/O from the Pi. It can deliver up to 1 A of current to each of two motors. V_{motor} should come from a separate battery pack; the 5 V output of the Pi cannot supply enough current to drive most motors and the Pi could be damaged.

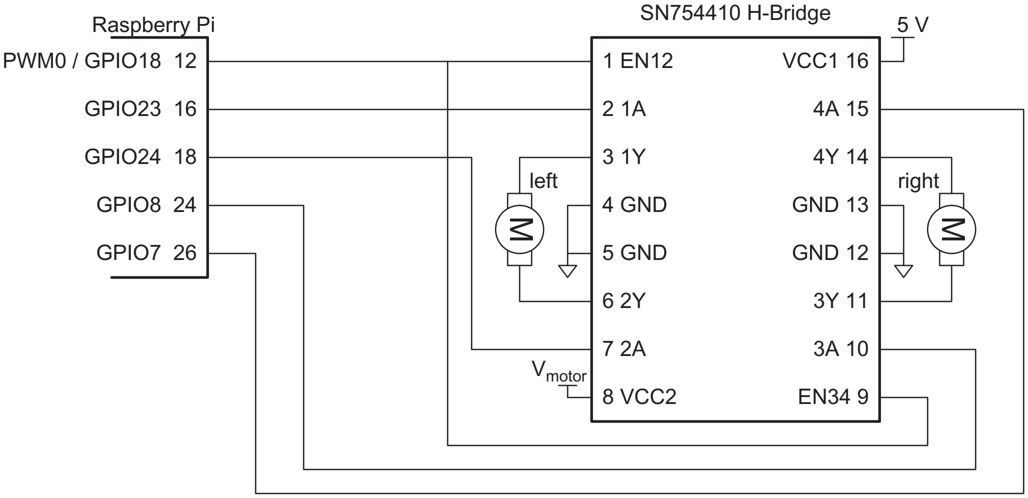

Raspberry Pi
PWM0 / GPIO18 12
GPIO23 16
GPIO24 18
GPIO8 24
GPIO7 26
SN754410 H-Bridge
1 EN12
2 1A
3 1Y
4 GND
5 GND
6 2Y
7 2A
8 VCC2
VCC1 16
4A 15
4Y 14
GND 13
GND 12
3Y 11
3A 10
EN34 9
5 V
left
right
M
M
V_motor

Figure e9.35 **Motor control with dual H-bridge**

Table e9.10 **H-Bridge control**

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

Table e9.10 describes how the inputs to each H-bridge control a motor. The microcontroller drives the enable signals with a PWM signal to control the speed of the motors. It drives the four other pins to control the direction of each motor.

The PWM is configured to work at about 5 kHz with a duty cycle ranging from 0 to 100%. Any PWM frequency far higher than the motor's bandwidth will give the effect of smooth movement. Note that the relationship between duty cycle and motor speed is nonlinear and that below some duty cycle, the motor will not move at all.

```
#include "EasyPIO.h"

// Motor Constants
#define MOTOR_1A 23
#define MOTOR_2A 24
#define MOTOR_3A 8
#define MOTOR_4A 7

void setSpeed(float dutycycle) {          // pwmInit() must be called first.
    setPWM(5000, dutycycle);
}

void setMotorLeft(int dir) {              // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir) {             // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1); // Both motors drive forward
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0); // Both motors drive backward
}

void left(void) {
    setMotorLeft(0); setMotorRight(1); // Left back, right forward

}
```

```
void right(void) {
    setMotorLeft(1); setMotorRight(0); // Right back, left forward
}

void halt(void) {                      // Turn both motors off
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                            // Ensure motors are not spinning
    pwmInit();                         // Turn on PWM
    setSpeed(0.75);                    // Default to partial power
}

main(void) {
    pioInit();
    initMotors();
    forward(); delayMillis(5000);
    backward(); delayMillis(5000);
    left(); delayMillis(5000);
    right(); delayMillis(5000);
    halt();
}
```

In the previous example, there is no way to measure the position of each motor. Two motors are unlikely to be exactly matched, so one is likely to turn slightly faster than the other, causing the robot to veer off course. To solve this problem, some systems add shaft encoders. Figure e9.36(a) shows a simple shaft encoder consisting of a disk with slots attached to the motor shaft. An LED is placed on one side and a light sensor is placed on the other side. The shaft encoder produces a pulse every time the gap rotates past the LED. A microcontroller can count these pulses to measure the total angle that the shaft has turned. By using two LED/sensor pairs spaced half a slot width apart, an improved shaft encoder can produce quadrature outputs shown in Figure e9.36(b) that indicate the direction the shaft is turning as well as the angle by which it has turned. Sometimes shaft encoders add another hole to indicate when the shaft is at an index position.

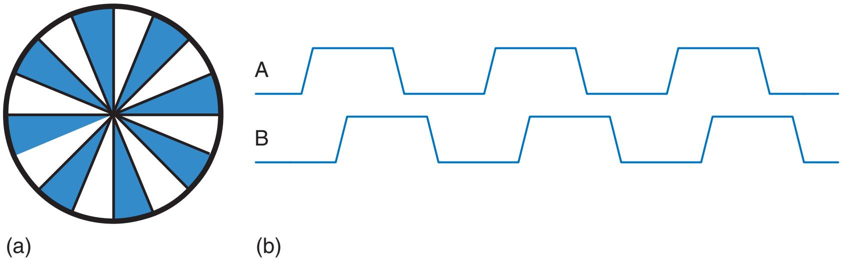


Figure e9.36 **Shaft encoder (a) disk, (b) quadrature outputs**

9.4.4.2 Servo Motor

A servo motor is a DC motor integrated with a gear train, a shaft encoder, and some control logic so that it is easier to use. They have a limited rotation, typically 180°. Figure e9.37 shows a servo with the lid removed to reveal the gears. A servo motor has a 3-pin interface with power (typically 5 V), ground, and a control input. The control input is typically a 50 Hz pulse-width modulated signal. The servo's control logic drives the shaft to a position determined by the duty cycle of the control input. The servo's shaft encoder is typically a rotary potentiometer that produces a voltage dependent on the shaft position.

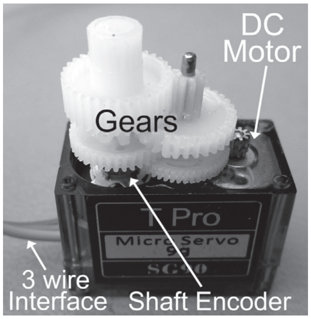


Figure e9.37 **SG90 servo motor**

In a typical servo motor with 180 degrees of rotation, a pulse width of 0.5 ms drives the shaft to 0°, 1.5 ms to 90°, and 2.5 ms to 180°. For example, Figure e9.38 shows a control signal with a 1.5 ms pulse width. Driving the servo outside its range may cause it to hit mechanical stops and be damaged. The servo's power comes from the power pin rather than the control pin, so the control can connect directly to a microcontroller without an H-bridge. Servo motors are commonly used in remote-control model airplanes and small robots because they are small, light, and convenient. Finding a motor with an adequate datasheet can be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

Example e9.13 SERVO MOTOR

Design a system in which a Raspberry Pi drives a servo motor to a desired angle.

Solution: Figure e9.39 shows a diagram of the connection to an SG90 servo motor, including the colors of the wires on the servo cable. The servo operates off of a 4.0–7.2 V power supply. It can draw as much as 0.5 A if it must deliver a large amount of force, but may run directly off the Raspberry Pi power supply if the load is light. A single wire carries the PWM signal, which can be provided at 5 or 3.3 V logic levels. The code configures the PWM generation and computes

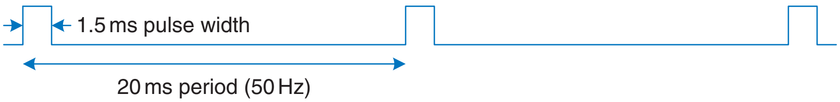


Figure e9.38 **Servo control waveform**

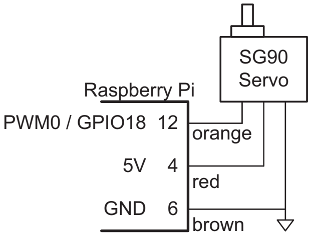


Figure e9.39 **Servo motor control**

the appropriate duty cycle for the desired angle. It cycles through positioning the servo at 0, 90, and 180 degrees.

```
#include "EasyPIO.h"
void setServo(float angle) {
  setPWM(50.0, 0.025 + (0.1 * (angle / 180)));
}

void main(void) {
  pioInit();
  pwmInit();
  while (1) {
    setServo(0.0);      // Left
    delayMillis(1000);
    setServo(90.0);     // Center
    delayMillis(1000);
    setServo(180.0);    // Right
    delayMillis(1000);
  }
}
```

It is also possible to convert an ordinary servo into a continuous rotation servo by carefully disassembling it, removing the mechanical stop, and replacing the potentiometer with a fixed voltage divider. Many websites show detailed directions for particular servos. The PWM will then control the velocity rather than position, with 1.5 ms indicating stop, 2.5 ms indicating full speed forward, and 0.5 ms indicating full speed backward. A continuous rotation servo may be more convenient and less expensive than a simple DC motor combined with an H-bridge and gear train.

9.4.4.3 Stepper Motor

A stepper motor advances in discrete steps as pulses are applied to alternate inputs. The step size is usually a few degrees, allowing precise positioning and continuous rotation. Small stepper motors generally come with two sets of coils called *phases* wired in *bipolar* or *unipolar* fashion. Bipolar motors are more powerful and less expensive for a given size but require an H-bridge driver, while unipolar motors can be driven with transistors acting as switches. This section focuses on the more efficient bipolar stepper motor.

Figure e9.40(a) shows a simplified two-phase bipolar motor with a 90° step size. The rotor is a permanent magnet with one north and one south pole. The stator is an electromagnet with two pairs of coils comprising

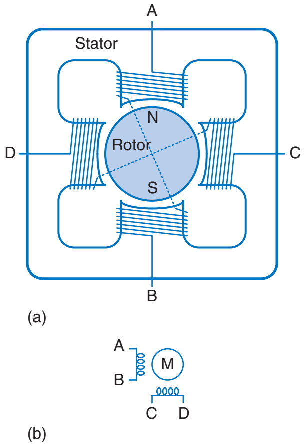
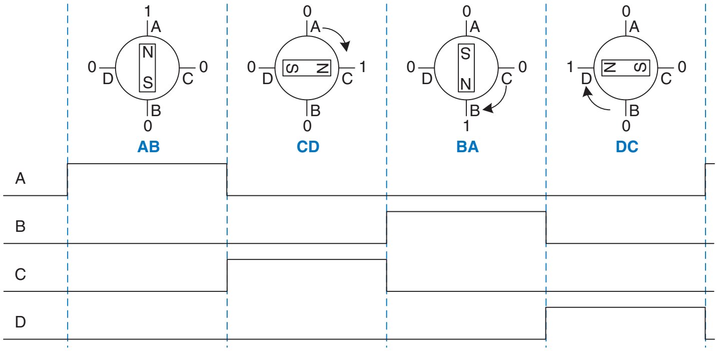
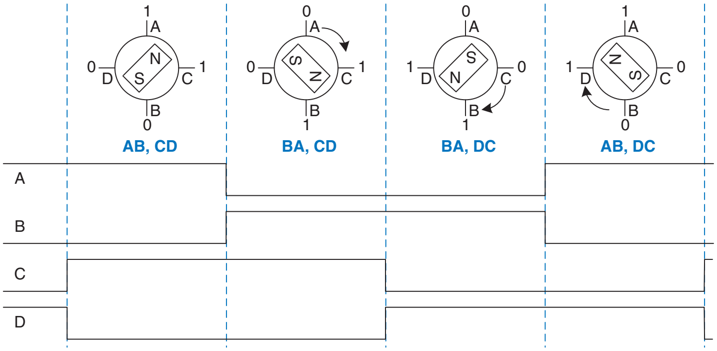

A
Stator
N
Rotor
S
D
C
B
(a)
A
B
M
C
D
(b)

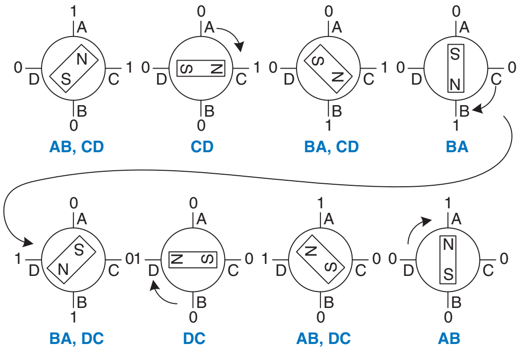
Figure e9.40 **Two-phase bipolar motor: (a) simplified diagram, (b) symbol**


AB
CD
BA
DC
A
B
C
D

(a) Wave drive


AB, CD
BA, CD
BA, DC
AB, DC
A
B
C
D

(b) Two-phase on drive


AB, CD
CD
BA, CD
BA
BA, DC
DC
AB, DC
AB

(c) Half-step drive

Figure e9.41 **Bipolar motor drive**

the two phases. Two-phase bipolar motors thus have four terminals. Figure e9.40(b) shows a symbol for the stepper motor modeling the two coils as inductors. Practical motors add gearing to reduce the output step size and increase torque.

Figure e9.41 shows three common drive sequences for a two phase bipolar motor. Figure e9.41(a) illustrates *wave drive*, in which the coils are energized in the sequence AB – CD – BA – DC. Note that BA means that the winding AB is energized with current flowing in the opposite direction; this is the origin of the name bipolar. The rotor turns by 90 degrees at each step. Figure e9.41(b) illustrates *two-phase-on drive*, following the pattern (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) indicates that both coils AB and CD are energized simultaneously. The rotor again turns by 90 degrees at each step, but aligns itself halfway between the two pole positions. This gives the highest torque operation because both coils are delivering power at once. Figure e9.41(c) illustrates *half-step drive*, following the pattern (AB, CD) – CD – (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. The rotor turns by 45 degrees at each half-step. The rate at which the pattern advances determines the speed of the motor. To reverse the motor direction, the same drive sequences are applied in the opposite order.

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure e9.42 shows an AIRPAX LB82773-M1 bipolar stepper motor with a 7.5 degree step size. The motor operates off 5 V and draws 0.8 A through each coil.

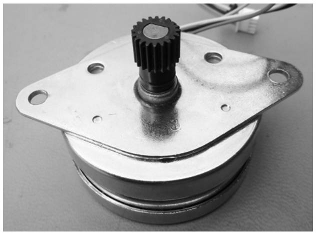


Figure e9.42 AIRPAX LB82773-M1 bipolar stepper motor

The torque in the motor is proportional to the coil current. This current is determined by the voltage applied and by the inductance L and resistance R of the coil. The simplest mode of operation is called *direct voltage drive* or *L/R drive*, in which the voltage V is directly applied to the coil. The current ramps up to $I = V/R$ with a time constant set by L/R, as shown in Figure e9.43(a). This works well for slow speed operation. However, at higher speed, the current doesn't have enough time to ramp up to the full level, as shown in Figure e9.43(b), and the torque drops off.

A more efficient way to drive a stepper motor is by pulse-width modulating a higher voltage. The high voltage causes the current to ramp up to full current more rapidly, then it is turned off (PWM) to avoid overloading the motor. The voltage is then modulated or *chopped* to maintain the current near the desired level. This is called *chopper constant current drive* and is shown in Figure e9.43(c). The controller uses a small resistor in series with the motor to sense the current being applied by measuring the voltage drop, and applies an enable signal to the H-bridge to turn off the drive when the current reaches the desired level. In principle, a microcontroller could generate the right waveforms, but it is easier to use a stepper motor controller. The L297 controller from ST Microelectronics is a convenient choice, especially when coupled with the L298 dual H-bridge with current sensing pins and a 2 A peak power capability. Unfortunately, the L298 is not available in a DIP package so it is harder to breadboard. ST's application notes AN460 and AN470 are valuable references for stepper motor designers.

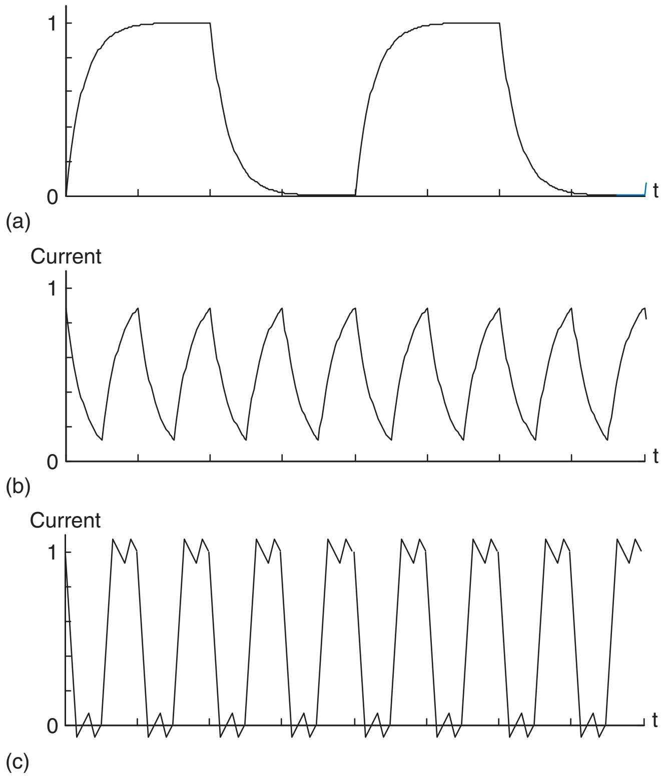


Figure e9.43 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

Example e9.14 BIPOLAR STEPPER MOTOR DIRECT WAVE DRIVE

Design a system to drive an AIRPAX bipolar stepper motor at a specified speed and direction using direct wave drive.

Solution: Figure e9.44 shows the bipolar stepper motor driven directly by an H-bridge with the same interface as the DC motor. Note that VCC2 must supply enough voltage and current to meet the motor's demands or else the motor may skip steps as the rotation rate increases.

```
#include "EasyPIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MICROS_PER_SEC 1000000
#define DEG_PER_REV 360

int stepperPins[] = {18, 8, 7, 23, 24};
```

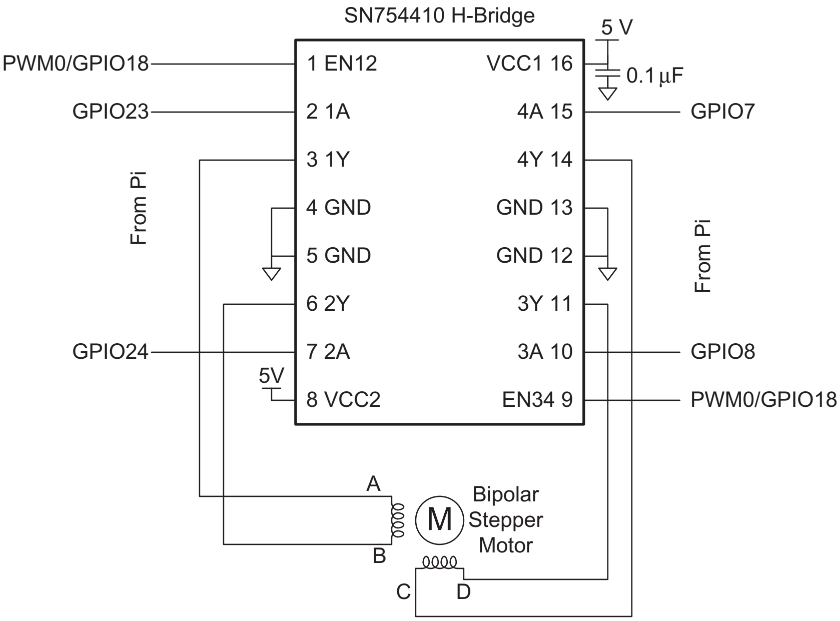

SN754410 H-Bridge
PWM0/GPIO18
GPIO23
From Pi
GPIO24
1 EN12
2 1A
3 1Y
4 GND
5 GND
6 2Y
7 2A
5V
8 VCC2
VCC1 16
4A 15
4Y 14
GND 13
GND 12
3Y 11
3A 10
EN34 9
5 V
0.1 μF
GPIO7
From Pi
GPIO8
PWM0/GPIO18
A
B
M
Bipolar Stepper Motor
C
D

Figure e9.44 Bipolar stepper motor direct drive with H-bridge

```
int curStepState;  // Keep track of the current position of stepper motor

void stepperInit(void) {
    pinsMode(stepperPins, 5, OUTPUT);
    curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
    int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001};    //{2A, 1A, 4A, 3A, EN}
    int step = 0;
    unsigned int microsPerStep = (SECS_PER_MIN * MICROS_PER_SEC * STEPSIZE) /
                                 (rpm * DEG_PER_REV);
    for (step = 0; step < steps; step++) {
        digitalWrites(stepperPins, 5, sequence[curStepState]);
        if (dir==0) curStepState = (curStepState + 1) % 4;
        else        curStepState = (curStepState + 3) % 4;
        delayMicros(microsPerStep);
    }
}

void main(void) {
    pioInit();
    stepperInit();
    stepperSpin(1, 12000, 120);  // Spin 60 revolutions at 120 rpm
}
```

9.5 BUS INTERFACES

A *bus interface* connects processors to memory and/or peripherals. In general, a bus interface supports one or more *bus masters* that can initiate read or write requests to the bus and one or more *slaves* that respond to the requests; processors are normally masters and memory and peripherals are slaves.

The *Advanced Microcontroller Bus Architecture* (AMBA) is an open standard bus interface for connecting components on a chip. Introduced by ARM in 1996, it has developed through multiple revisions to boost performance and features and has become a *de facto* standard for embedded microcontrollers. The *Advanced High-performance Bus* (AHB) is one of the AMBA standards. AHB-Lite is a simplified version of AHB that supports a single bus master. This section describes AHB-Lite to illustrate the characteristics of a typical bus interface and to show how to design memory and peripherals that interface to a standard bus.

AHB is an example of a point-to-point read bus, in contrast with older bus architectures that use a single shared data bus where each slave accesses the bus via a tristate driver. Using point-to-point links between each slave and the read multiplexer allows the bus to run faster and avoids wasting power when one slave turns on its driver before another has turned off.

9.5.1 AHB-Lite

Figure e9.45 shows a simple AHB-Lite bus connecting a processor (bus master) to RAM, ROM, and two peripherals (slaves). Observe that the bus is very similar to the one from Figure e9.1 except that the names have changed. The master provides a synchronous clock (HCLK) to all of the slaves and can reset the slaves by asserting HRESETn low. The master sends an address. The address decoder uses the most significant bits to generate the HSEL signal selecting which slave to access, and the slaves use the least significant bits to define the memory location or register. The master sends HWDATA for writes. Each slave reads onto its own HRDATA, and a multiplexer chooses the data from the selected slave.

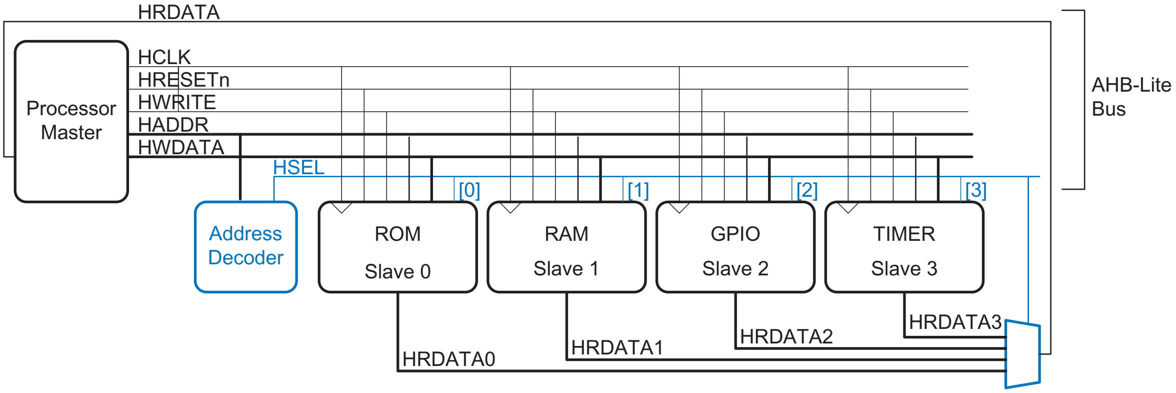


Figure e9.45 AHB-Lite bus

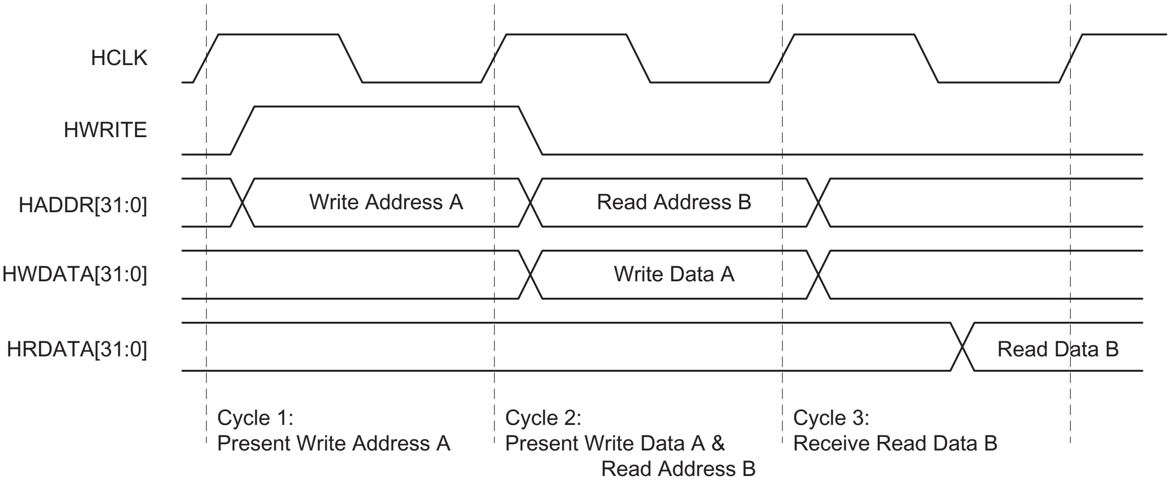

HCLK
HWRITE
HADDR[31:0]
HWDATA[31:0]
HRDATA[31:0]
Write Address A
Read Address B
Write Data A
Read Data B
Cycle 1:
Present Write Address A
Cycle 2:
Present Write Data A &
Read Address B
Cycle 3:
Receive Read Data B

Figure e9.46 **AHB-Lite transfer timing**

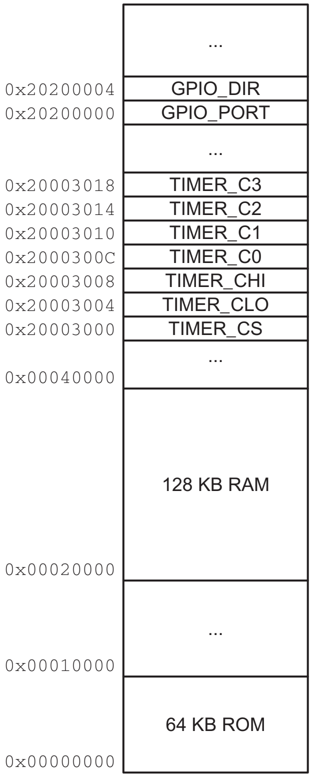

...
0x20200004
GPIO_DIR
0x20200000
GPIO_PORT
...
0x20003018
TIMER_C3
0x20003014
TIMER_C2
0x20003010
TIMER_C1
0x2000300C
TIMER_C0
0x20003008
TIMER_CHI
0x20003004
TIMER_CLO
0x20003000
TIMER_CS
...
0x00040000
128 KB RAM
0x00020000
...
0x00010000
64 KB ROM
0x00000000

Figure e9.47 **System memory map**

The master sends a 32-bit address on one cycle and writes or reads data on the subsequent cycle. The write or read is called a *transfer*. For writes, the master raises HWRITE and sends the 32-bit HWDATA to write. For reads, the master lowers HWRITE and the slave responds with 32-bit HRDATA. Transfers can overlap so that the master can send the address of the next transfer while reading or writing data for the current transfer. Figure e9.46 illustrates the timing of the bus for a write followed immediately by a read. Observe how the data lags one cycle behind the address and how the two transfers partially overlap.

In this example, we assume the bus transfers a single 32-bit word at a time and that the slave responds in one clock cycle. AHB-Lite defines additional signals to specify the size of the transfer (8 – 1024 bits) and to transfer bursts of 4 to 16 elements. The master can also specify types of transfers, protection, and bus locking. Slaves can deassert HREADY to indicate that they need multiple clock cycles to respond, or can assert HRESP to indicate an error. Interested readers should consult the *AMBA 3 AHB-Lite Protocol Specification*, available online.

9.5.2 Memory and Peripheral Interface Example

This section illustrates connecting RAM, ROM, GPIO, and a timer to a processor over an AHB-Lite bus. Figure e9.47 shows a memory map for the system from Figure e9.45 with 128 KB of RAM and 64 KB of ROM. The GPIO controls 32 I/O pins. The 32-bit GPIO_DIR register controls whether each pin is an output (1) or an input (0). The 32-bit

GPIO_PORT register can be written to specify the value of outputs and read to return the values on the pins. The Timer module resembles the BCM2835 counter described in Section 9.3.5, containing a 64-bit counter running at the HCLK frequency (TIMER_CHI:TIMER_CLO), four 32-bit compare channels (TIMER_C3:0), and a match register (TIMER_CS).

HDL Example e9.1 lists SystemVerilog code for the system. The decoder is based on the memory map. The memories and peripherals interface to the bus. Unnecessary signals are omitted; for example, the ROM ignores writes. The GPIO module also connects to 32 I/O pins that can behave as inputs or outputs.

HDL Example e9.1

```
module ahb_lite(input logic          HCLK,
                input logic          HRESETn,
                input logic  [31:0]  HADDR,
                input logic          HWRITE,
                input logic  [31:0]  HWDATA,
                output logic [31:0]  HRDATA,
                inout tri    [31:0]  pins);

  logic [3:0] HSEL;
  logic [31:0] HRDATA0,  HRDATA1,  HRDATA2, HRDATA3;
  logic [31:0] pins_dir, pins_out, pins_in;
  logic [31:0] HADDRDEL;
  logic HWRITEDEL;

  // Delay address and write signals to align in time with data
  flop #(32) adrreg(HCLK, HADDR, HADDRDEL);
  flop #(1)       writereg(HCLK, HWRITE, HWRITEDEL);

  // Memory map decoding
  ahb_decoder dec(HADDRDEL, HSEL);
  ahb_mux   mux(HSEL,  HRDATA0,  HRDATA1,  HRDATA2,  HRDATA3,
                HRDATA);

  // Memory and peripherals
  ahb_rom     rom (HCLK, HSEL[0], HADDRDEL[15:2], HRDATA0);
  ahb_ram     ram (HCLK, HSEL[1], HADDRDEL[16:2], HWRITEDEL,
              HWDATA, HRDATA1);
  ahb_gpio    gpio (HCLK, HRESETn, HSEL[2], HADDRDEL[2],
                     HWRITEDEL, HWDATA, HRDATA2, pins);
  ahb_timer   timer(HCLK, HRESETn, HSEL[3], HADDRDEL[4:2],
                     HWRITEDEL, HWDATA, HRDATA3);
endmodule

module ahb_decoder(input logic [31:0] HADDR,
                   output logic [3:0] HSEL);

  // Decode based on most significant bits of the address
  assign HSEL[0]=(HADDR[31:16]
              ==16'h0000);  //  64KB  ROM  at  0x00000000  -
                                 0x0000FFFF
  assign HSEL[1]=(HADDR[31:17]
              ==15'h0001);  //  128KB  RAM  at  0x00020000  -
                                 0x003FFFFF
  assign HSEL[2]=(HADDR[31:4]
              ==28'h2020000);  //  GPIO  at  0x20200000  -
              0x20200007
  assign HSEL[3]=(HADDR[31:8]
              ==24'h200030);  //   Timer   at   0x20003000   -
                                  0x2000301B
endmodule

module ahb_mux(input  logic [3:0]  HSEL,
               input  logic [31:0] HRDATA0, HRDATA1, HRDATA2,
               HRDATA3,
               output logic [31:0] HRDATA);

  always_comb
    casez(HSEL)
      4'b???1: HRDATA <= HRDATA0;
      4'b??10: HRDATA <= HRDATA1;
      4'b?100: HRDATA <= HRDATA2;
      4'b1000: HRDATA <= HRDATA3;
    endcase
endmodule

module ahb_ram(input  logic        HCLK,
               input  logic        HSEL,
               input  logic [16:2] HADDR,
               input  logic        HWRITE,
               input  logic [31:0] HWDATA,
               output logic [31:0] HRDATA);

  logic [31:0] ram[32767:0]; // 128KB RAM organized as 32K
                              x 32 bits

  assign HRDATA = ram[HADDR]; // *** check addressing is
                                correct
  always_ff @(posedge HCLK)
    if (HWRITE & HSEL) ram[HADDR] <= HWDATA;
endmodule

module ahb_rom(input  logic        HCLK,
               input  logic        HSEL,
               input  logic [16:2] HADDR,
               output logic [31:0] HRDATA);
  logic [31:0] rom[16383:0]; // 64KB ROM organized as 16K x
                              32 bits

  // *** load ROM from disk file

  assign  HRDATA  =  rom[HADDR];  //  ***  check  addressing  is
                                       correct

endmodule
```

```
module ahb_gpio(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic        HSEL,
                input  logic [2]    HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                output logic [31:0] pin_dir,
                output logic [31:0] pin_out,
                input  logic [31:0] pin_in);

  logic [31:0] gpio[1:0];  // GPIO registers

  // write selected register
  always_ff @(posedge HCLK or negedge HRESETn)
    if (~HRESETn) begin
      gpio[0] <= 32'b0;  // GPIO_PORT
      gpio[1] <= 32'b0;  // GPIO_DIR
    end else if (HWRITE & HSEL)
      gpio[HADDR] <= HWDATA;

  // read selected register
  assign HRDATA = HADDR ? gpio[1] : pin_in;

  // send value and direction to I/O drivers
  assign pin_out = gpio[0];
  assign pin_dir = gpio[1];
endmodule

module ahb_timer(input  logic        HCLK,
                 input  logic        HRESETn,
                 input  logic        HSEL,
                 input  logic [4:2]  HADDR,
                 input  logic        HWRITE,
                 input  logic [31:0] HWDATA,
                 output logic [31:0] HRDATA);

  logic [31:0] timers[6:0]; // timer registers
  logic [31:0] chi, clo;    // next counter value
  logic [3:0]  match, clr;  // determine if counter matches
                               compare reg

  // write selected register and update tiers and match
    always_ff @(posedge HCLK or negedge HRESETn)
  if (~HRESETn) begin
    timers[0] <= 32'b0;  // TIMER_CS
    timers[1] <= 32'b0;  // TIMER_CLO
    timers[2] <= 32'b0;  // TIMER_CHI
    timers[3] <= 32'b0;  // TIMER_C0
    timers[4] <= 32'b0;  // TIMER_C1
    timers[5] <= 32'b0;  // TIMER_C2
    timers[6] <= 32'b0;  // TIMER_C3
  end else begin
    timers[0] <= {28'b0, match};
    timers[1] <= (HWRITE & HSEL & HADDR = = 3'b000) ?
                   HWDATA : clo
    timers[2] <= (HWRITE & HSEL & HADDR = = 3'b000) ?
                   HWDATA : chi;
    if (HWRITE & HSEL & HADDR = = 3'b011) timers[3] <= HWDATA;
    if (HWRITE & HSEL & HADDR = = 3'b100) timers[4] <= HWDATA;
    if (HWRITE & HSEL & HADDR = = 3'b101) timers[5] <= HWDATA;
    if (HWRITE & HSEL & HADDR = = 3'b110) timers[6] <= HWDATA;
  end

  // read selected register
  assign HRDATA = timers[HADDR];

  // increment 64-bit counter as pair of TIMER_CHI, TIMER_CLO
  assign {chi, clo} = {timers[2], timers[1]} + 1;

  // generate matches: set match bit when counter matches
     compare register
  //  clear bit when a 1 is written to that position of the match
      register
  assign clr = (HWRITE & HSEL & HADDR = = 3'b000 & HWDATA[3:0]);
  assign match[0] = ~clr[0] & (timers[0][0] |
                    (timers[1] = = timers[3]));
  assign match[1] = ~clr[1] & (timers[0][1] |
                    (timers[1] = = timers[4]));
  assign match[2] = ~clr[2] & (timers[0][2] |
                    (timers[1] = = timers[5]));
  assign match[3] = ~clr[3] & (timers[0][3] |
                    (timers[1] = = timers[6]));
endmodule

module gpio_pins(input  logic [31:0] pin_dir, // 1 = output,
                                                 0 = input
                 input  logic [31:0] pin_out, // value to drive
                                                 on outputs
                 output logic [31:0] pin_in,  // value read
                                                 from pins
                 inout  tri   [31:0] pin);   // tristate pins

  // Individual tristate control of each pin

  // No graceful way to control tristates on a per-bit basis in
     SystemVerilog
  genvar i;
  generate
  for (i=0; i<32; i=i+1) begin: pinloop
    assign pin[i] = pin_dir[i] ? pin_out[i] : 1'bz;
  end
  endgenerate

  assign pin_in = pin;
endmodule
```

9.6 PC I/O SYSTEMS

Personal computers (PCs) use a wide variety of I/O protocols for purposes including memory, disks, networking, internal expansion cards, and external devices. These I/O standards have evolved to offer very high performance and to make it easy for users to add devices. These attributes come at the expense of complexity in the I/O protocols. This section explores the major I/O standards used in PCs and examines some options for connecting a PC to custom digital logic or other external hardware.

Figure e9.48 shows a PC motherboard for a Core i5 or i7 processor. The processor is packaged in a *land grid array* with 1156 gold-plated pads to supply power and ground to the processor and connect the processor to memory and I/O devices. The motherboard contains the DRAM memory module slots, a wide variety of I/O device connectors, and the power supply connector, voltage regulators, and capacitors. A pair of DRAM modules are connected over a DDR3 interface. External peripherals such as keyboards or webcams are attached over USB. High-performance expansion cards such as graphics cards connect over the PCI Express x16 slot, while lower-performance cards can use PCI Express x1 or the older PCI slots. The PC connects to the network using the Ethernet jack. The hard disk connects to a SATA port. The remainder of this section gives an overview of the operation of each of these I/O standards.

One of the major advances in PC I/O standards has been the development of high-speed serial links. Until recently, most I/O was built around parallel links consisting of a wide data bus and a clock signal. As data rates increased, the difference in delay among the wires in the bus set a limit to how fast the bus could run. Moreover, busses connected to multiple devices suffer from transmission line problems such as reflections and different flight times to different loads. Noise can also corrupt the data. Point-to-point serial

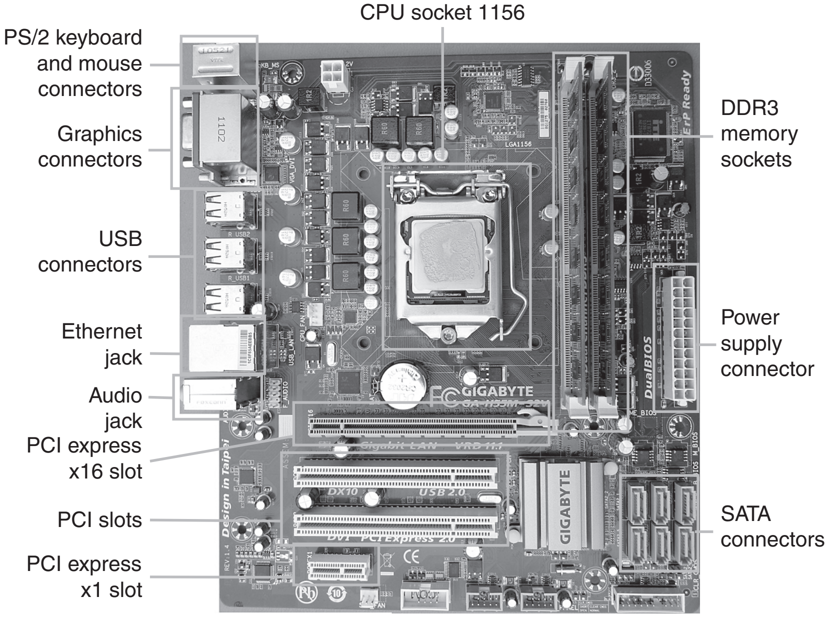

CPU socket 1156
PS/2 keyboard and mouse connectors
Graphics connectors
USB connectors
Ethernet jack
Audio jack
PCI express x16 slot
PCI slots
PCI express x1 slot
DDR3 memory sockets
Power supply connector
SATA connectors

Figure e9.48 **Gigabyte GA-H55M-S2V motherboard**

links eliminate many of these problems. The data is usually transmitted on a differential pair of wires. External noise that affects both wires in the pair equally is unimportant. The transmission lines are easy to properly terminate, so reflections are small (see Section A.8 on transmission lines). No explicit clock is sent; instead, the clock is recovered at the receiver by watching the timing of the data transitions. High-speed serial link design is a specialized subject, but good links can run faster than 10 Gb/s over copper wires and even faster along optical fibers.

9.6.1 USB

Until the mid-1990s, adding a peripheral to a PC took some technical savvy. Adding expansion cards required opening the case, setting jumpers to the correct position, and manually installing a device driver. Adding an RS-232 device required choosing the right cable and properly configuring the baud rate, and data, parity, and stop bits. The *Universal Serial Bus* (USB), developed by Intel, IBM, Microsoft, and others, greatly simplified adding peripherals by standardizing the cables and software configuration process. Billions of USB peripherals are now sold each year.

USB 1.0 was released in 1996. It uses a simple cable with four wires: 5 V, GND, and a differential pair of wires to carry data. The cable is impossible to plug in backward or upside down. It operates at up to 12 Mb/s. A device can pull up to 500 mA from the USB port, so keyboards, mice, and other peripherals can get their power from the port rather than from batteries or a separate power cable.

USB 2.0, released in 2000, upgraded the speed to 480 Mb/s by running the differential wires much faster. With the faster link, USB became practical for attaching webcams and external hard disks. Flash memory sticks with a USB interface also replaced floppy disks as a means of transferring files between computers.

USB 3.0, released in 2008, further boosted the speed to 5 Gb/s. It uses the same shape connector, but the cable has more wires that operate at very high speed. It is well suited to connecting high-performance hard disks. At about the same time, USB added a Battery Charging Specification that boosts the power supplied over the port to speed up charging mobile devices.

The simplicity for the user comes at the expense of a much more complex hardware and software implementation. Building a USB interface from the ground up is a major undertaking. Even writing a simple device driver is moderately complex.

9.6.2 PCI and PCI Express

The *Peripheral Component Interconnect* (PCI) bus is an expansion bus standard developed by Intel that became widespread around 1994. It was used to add expansion cards such as extra serial or USB ports, network interfaces, sound cards, modems, disk controllers, or video cards. The 32-bit parallel bus operates at 33 MHz, giving a bandwidth of 133 MB/s.

The demand for PCI expansion cards has steadily declined. More standard ports such as Ethernet and SATA are now integrated into the motherboard. Many devices that once required an expansion card can now be connected over a fast USB 2.0 or 3.0 link. And video cards now require far more bandwidth than PCI can supply.

Contemporary motherboards often still have a small number of PCI slots, but fast devices like video cards are now connected via *PCI Express* (PCIe). PCI Express slots provide one or more lanes of high-speed serial links. In PCIe 3.0, each lane operates at up to 8 Gb/s. Most motherboards provide an x16 slot with 16 lanes giving a total of 16 GB/s of bandwidth to data-hungry devices such as video cards.

9.6.3 DDR3 Memory

DRAM connects to the microprocessor over a parallel bus. In 2015, the present standard is DDR3, a third generation of double-data rate memory bus operating at 1.5 V. Typical motherboards now come with two DDR3 channels so they can simultaneously access two banks of memory modules. DDR4 is emerging, operating at 1.2V and higher speed.

Figure e9.49 shows a 4 GB DDR3 dual inline memory module (DIMM). The module has 120 contacts on each side, for a total of 240 connections, including a 64-bit data bus, a 16-bit time-multiplexed address bus, control signals, and numerous power and ground pins. In 2015, DIMMs typically carry 1–16 GB of DRAM. Memory capacity has been doubling approximately every 2–3 years.

DRAM presently operates at a clock rate of 100–266 MHz. DDR3 operates the memory bus at four times the DRAM clock rate. Moreover, it transfers data on both the rising and falling edges of the clock. Hence, it sends 8 words of data for each memory clock. At 64 bits/word, this corresponds to 6.4–17 GB/s of bandwidth. For example, DDR3-1600 uses a 200 MHz memory clock and an 800 MHz I/O clock to send 1600 million words/sec, or 12800 MB/s. Hence, the modules are

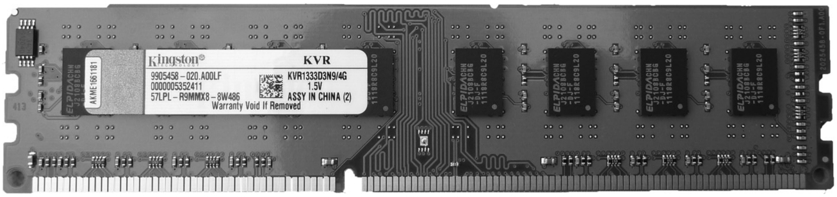


Figure e9.49 **DDR3 memory module**

also called PC3-12800. Unfortunately, DRAM latency remains high, with a roughly 50 ns lag from a read request until the arrival of the first word of data.

9.6.4 Networking

Computers connect to the Internet over a network interface running the *Transmission Control Protocol and Internet Protocol* (TCP/IP). The physical connection may be an Ethernet cable or a wireless Wi-Fi link.

Ethernet is defined by the IEEE 802.3 standard. It was developed at Xerox Palo Alto Research Center (PARC) in 1974. It originally operated at 10 Mb/s (called 10 Mbit Ethernet), but now is commonly found at 100 Mbit (Mb/s) and 1 Gbit (Gb/s) running on Category 5 cables containing four twisted pairs of wires. 10 Gbit Ethernet running on fiber optic cables is increasingly popular for servers and other high-performance computing, and 100 Gbit Ethernet is emerging.

Wi-Fi is the popular name for the IEEE 802.11 wireless network standard. It operates in the 2.4 and 5 GHz unlicensed wireless bands, meaning that the user doesn't need a radio operator's license to transmit in these bands at low power. Table e9.11 summarizes the capabilities of three generations of Wi-Fi; the emerging 802.11ac standard promises to push wireless data rates beyond 1 Gb/s. The increasing performance comes from advancing modulation and signal processing, multiple antennas, and wider signal bandwidths.

9.6.5 SATA

Internal hard disks require a fast interface to a PC. In 1986, Western Digital introduced the *Integrated Drive Electronics* (IDE) interface, which evolved into the *AT Attachment* (ATA) standard. The standard uses a bulky 40 or 80-wire ribbon cable with a maximum length of 18″ to send data at 16–133 MB/s.

Table e9.11 **802.11 Wi-Fi protocols**

Protocol	Release	Frequency Band (GHz)	Data Rate (Mb/s)	Range (m)
802.11b	1999	2.4	5.5–11	35
802.11g	2003	2.4	6–54	38
802.11n	2009	2.4/5	7.2–150	70
802.11ac	2013	5	433+	variable

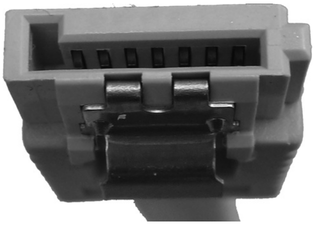


Figure e9.50 **SATA cable**

ATA has been supplanted by Serial ATA (SATA), which uses high-speed serial links to run at 1.5, 3, or 6 Gb/s over a more convenient 7-conductor cable shown in Figure e9.50. The fastest solid-state drives in 2015 exceed 500 MB/s of bandwidth, taking full advantage of SATA.

A related standard is Serial Attached SCSI (SAS), an evolution of the parallel SCSI (Small Computer System Interface). SAS offers performance comparable to SATA and supports longer cables; it is common in server computers.

9.6.6 Interfacing to a PC

All of the PC I/O standards described so far are optimized for high performance and ease of attachment but are difficult to implement in hardware. Engineers and scientists often need a way to connect a PC to external circuitry, such as sensors, actuators, microcontrollers, or FPGAs. The serial connection described in Section 9.3.4.2 is sufficient for a low-speed connection to a microcontroller with a UART. This section describes two more means: data acquisition systems, and USB links.

9.6.6.1 Data Acquisition Systems

Data Acquisition Systems (DAQs) connect a computer to the real world using multiple channels of analog and/or digital I/O. DAQs are now commonly available as USB devices, making them easy to install. National Instruments (NI) is a leading DAQ manufacturer.

High-performance DAQ prices tend to run into the thousands of dollars, mostly because the market is small and has limited competition. Fortunately, NI sells their handy myDAQ system at a student discount price of $200 including their LabVIEW software. Figure e9.51 shows a myDAQ. It has two analog channels capable of input and output

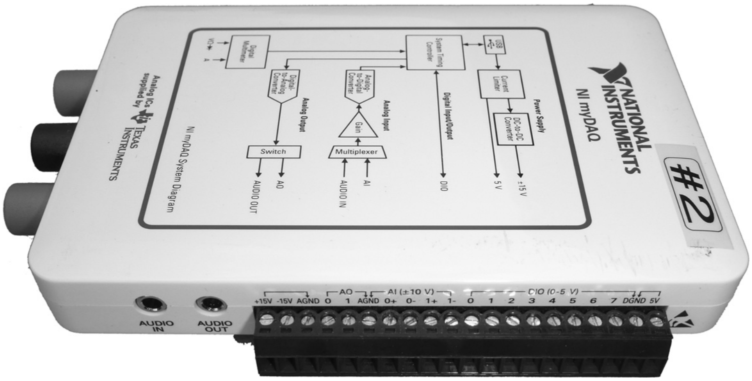

NATIONAL INSTRUMENTS
NI myDAQ
#2
Digital Multimeter
V/Ω
A
System Timing Controller
USB
Current Limiter
DC-to-DC Converter
Power Supply
Digital-to-Analog Converter
Analog Output
Switch
AO
AUDIO OUT
Analog-to-Digital Converter
Gain
Analog Input
Multiplexer
AI
AUDIO IN
Digital Input/Output
DIO
5 V
±15 V
NI myDAQ System Diagram
Analog ICs supplied by TEXAS INSTRUMENTS
AUDIO IN
AUDIO OUT
AO
AI (±10 V)
DIO (0-5 V)
+15V -15V AGND 0 1 AGND 0+ 0- 1+ 1- 0 1 2 3 4 5 6 7 DGND 5V

Figure e9.51 **NI myDAQ**

at 200 ksamples/sec with a 16-bit resolution and ±10 V dynamic range. These channels can be configured to operate as an oscilloscope and signal generator. It also has eight digital input and output lines compatible with 3.3 and 5 V systems. Moreover, it generates 5, 15, and −15 V power supply outputs and includes a digital multimeter capable of measuring voltage, current, and resistance. Thus, the myDAQ can replace an entire bench of test and measurement equipment while simultaneously offering automated data logging.

Most NI DAQs are controlled with LabVIEW, NI's graphical language for designing measurement and control systems. Some DAQs can also be controlled from C programs using the LabWindows environment, from Microsoft .NET applications using the Measurement Studio environment, or from Matlab using the Data Acquisition Toolbox.

9.6.6.2 USB Links

An increasing variety of products now provide simple, inexpensive digital links between PCs and external hardware over USB. These products contain predeveloped drivers and libraries, allowing the user to easily write a program on the PC that blasts data to and from an FPGA or microcontroller.

FTDI is a leading vendor for such systems. For example, the FTDI C232HM-DDHSL USB to Multi-Protocol Synchronous Serial Engine (MPSSE) cable shown in Figure e9.52 provides a USB jack at one end and, at the other end, an SPI interface operating at up to 30 Mb/s, along with 3.3 V power and four general purpose I/O pins. Figure e9.53 shows an example of connecting a PC to an FPGA using the cable. The cable can optionally supply 3.3 V power to the FPGA. The three SPI pins connect to an FPGA slave device like the one from Example e9.4. The figure also shows one of the GPIO pins used to drive an LED.

The PC requires the D2XX dynamically linked library driver to be installed. You can then write a C program using the library to send data over the cable.

Figure e9.52 **FTDI USB to MPSSE cable**

(© 2012 by FTDI; reprinted with permission.)

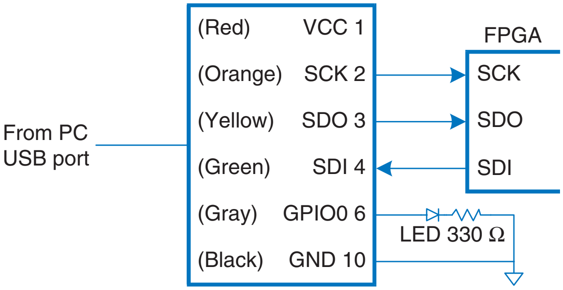


Figure e9.53 **C232HM-DDHSL USB to MPSESE interface from PC to FPGA**

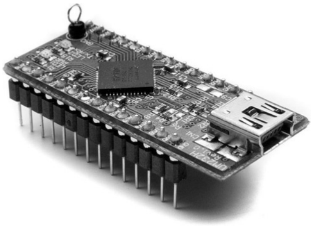


Figure e9.54 **FTDI UM232H module**
(© 2012 by FTDI; reprinted with permission.)

If an even faster connection is required, the FTDI UM232H module shown in Figure e9.54 links a PC's USB port to an 8-bit synchronous parallel interface operating up to 40 MB/s.

9.7 SUMMARY

Most processors use memory-mapped I/O to communicate with the real world. Microcontrollers offer a range of basic peripherals including general-purpose, serial, and analog I/O and timers. PCs and advanced microcontrollers support more complex I/O standards including USB, Ethernet, and SATA.

This chapter has provided many specific examples of I/O using the Raspberry Pi. Embedded system designers continually encounter new processors and peripherals. The general principal for simple embedded I/O is to consult the datasheet to identify the peripherals that are available and which pins and memory-mapped I/O registers are involved. Then it is usually straightforward to write a simple device driver that initializes the peripheral and then transmits or receives data.

For more complex standards such as USB, writing a device driver is a highly specialized undertaking best done by an expert with detailed knowledge of the device and the USB protocol stack. Casual designers should select a processor that comes with proven device drivers and example code for the devices of interest.

This page intentionally left blank

Digital System Implementation

A

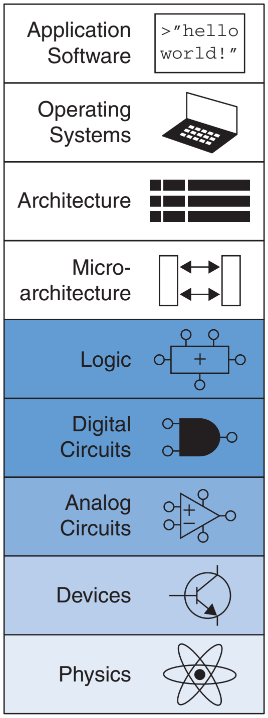
A.1 Introduction
A.2 74xx Logic
A.3 Programmable Logic
A.4 Application-Specific Integrated Circuits
A.5 Data sheets
A.6 Logic Families
A.7 Packaging and Assembly
A.8 Transmission Lines
A.9 Economics

A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book, however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

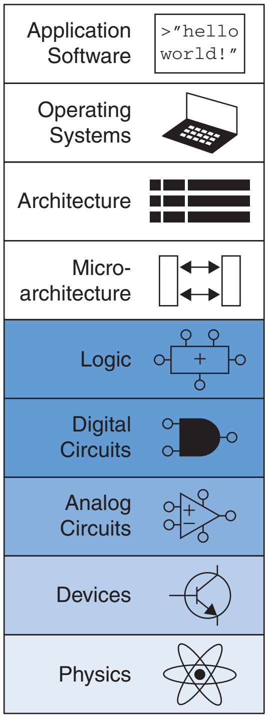
The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.



Digital System Implementation

eA

A.1 Introduction
A.2 74xx Logic
A.3 Programmable Logic
A.4 Application-Specific Integrated Circuits
A.5 Data Sheets
A.6 Logic Families
A.7 Packaging and Assembly
A.8 Transmission Lines
A.9 Economics



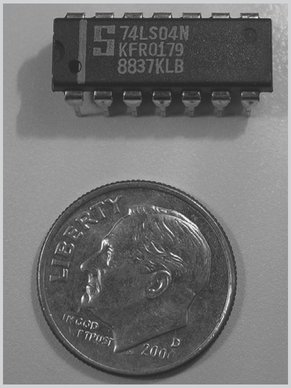
A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book, however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

A.2 74xx LOGIC

In the 1970s and 1980s, many digital systems were built from simple chips, each containing a handful of logic gates. For example, the 7404 chip contains six NOT gates, the 7408 contains four AND gates, and the 7474 contains two flip-flops. These chips are collectively referred to as *74xx-series* logic. They were sold by many manufacturers, typically for 10 to 25 cents per chip. These chips are now largely obsolete, but they are still handy for simple digital systems or class projects, because they are so inexpensive and easy to use. 74xx-series chips are commonly sold in 14-pin *dual inline packages* (*DIPs*).



74LS04 inverter chip in a 14-pin dual inline package. The part number is on the first line. LS indicates the logic family (see Section A.6). The N suffix indicates a DIP package. The large S is the logo of the manufacturer, Signetics. The bottom two lines of gibberish are codes indicating the batch in which the chip was manufactured.

A.2.1 Logic Gates

Figure eA.1 shows the pinout diagrams for a variety of popular 74xx-series chips containing basic logic gates. These are sometimes called *small-scale integration* (*SSI*) chips, because they are built from a few transistors. The 14-pin packages typically have a notch at the top or a dot on the top left to indicate orientation. Pins are numbered starting with 1 in the upper left and going counterclockwise around the package. The chips need to receive power ($V_{DD} = 5$ V) and ground (GND = 0 V) at pins 14 and 7, respectively. The number of logic gates on the chip is determined by the number of pins. Note that pins 3 and 11 of the 7421 chip are not connected (NC) to anything. The 7474 flip-flop has the usual D, CLK, and Q terminals. It also has a complementary output, $\overline{Q}$. Moreover, it receives asynchronous set (also called preset, or PRE) and reset (also called clear, or CLR) signals. These are active low; in other words, the flop sets when $\overline{PRE} = 0$, resets when $\overline{CLR} = 0$, and operates normally when $\overline{PRE} = \overline{CLR} = 1$.

A.2.2 Other Functions

The 74xx series also includes somewhat more complex logic functions, including those shown in Figures eA.2 and eA.3. These are called *medium-scale integration* (*MSI*) chips. Most use larger packages to accommodate more inputs and outputs. Power and ground are still provided at the upper right and lower left, respectively, of each chip. A general functional description is provided for each chip. See the manufacturer's data sheets for complete descriptions.

A.3 PROGRAMMABLE LOGIC

Programmable logic consists of arrays of circuitry that can be configured to perform specific logic functions. We have already introduced three forms of programmable logic: programmable read only memories (PROMs), programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). This section shows chip implementations for each of these. Configuration of these chips may be performed by blowing on-chip fuses to connect or disconnect circuit elements. This is called *one-time programmable* (*OTP*) logic because, once a fuse is blown, it cannot be restored. Alternatively, the configuration may be stored in a memory that can be reprogrammed at will. Reprogrammable logic is convenient in the laboratory because the same chip can be reused during development.

A.3.1 PROMs

As discussed in Section 5.5.7, PROMs can be used as lookup tables. A 2^N-word × M-bit PROM can be programmed to perform any combinational function of N inputs and M outputs. Design changes simply involve

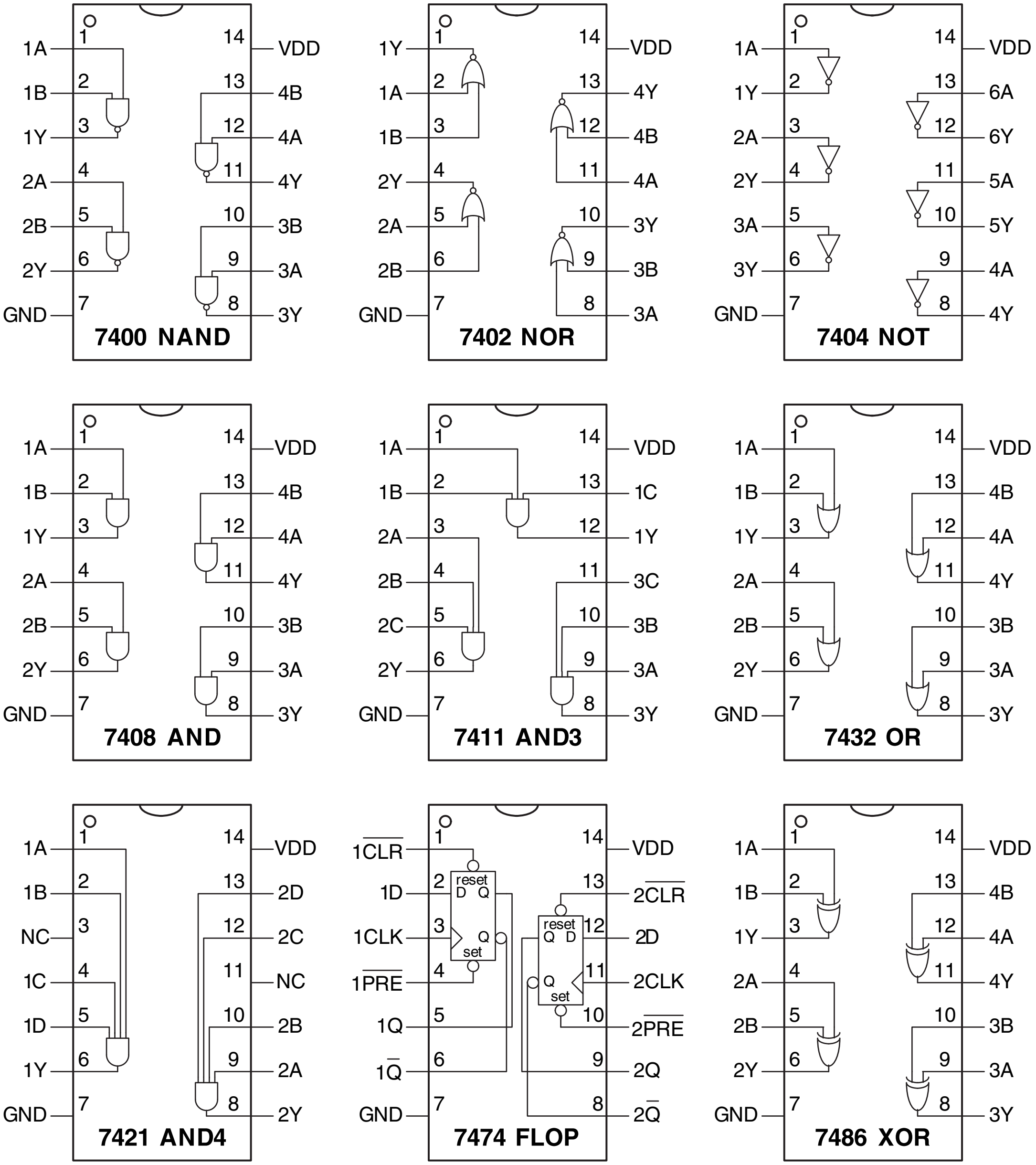
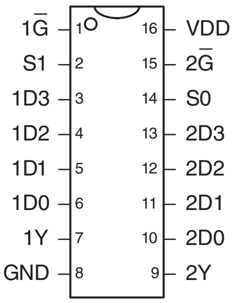

7400 NAND
1A
1B
1Y
2A
2B
2Y
GND
VDD
4B
4A
4Y
3B
3A
3Y
7402 NOR
1Y
1A
1B
2Y
2A
2B
GND
VDD
4Y
4B
4A
3Y
3B
3A
7404 NOT
1A
1Y
2A
2Y
3A
3Y
GND
VDD
6A
6Y
5A
5Y
4A
4Y
7408 AND
1A
1B
1Y
2A
2B
2Y
GND
VDD
4B
4A
4Y
3B
3A
3Y
7411 AND3
1A
1B
2A
2B
2C
2Y
GND
VDD
1C
1Y
3C
3B
3A
3Y
7432 OR
1A
1B
1Y
2A
2B
2Y
GND
VDD
4B
4A
4Y
3B
3A
3Y
7421 AND4
1A
1B
NC
1C
1D
1Y
GND
VDD
2D
2C
NC
2B
2A
2Y
7474 FLOP
1CLR
1D
1CLK
1PRE
1Q
1Q
GND
VDD
2CLR
2D
2CLK
2PRE
2Q
2Q
reset
D
Q
Q
set
7486 XOR
1A
1B
1Y
2A
2B
2Y
GND
VDD
4B
4A
4Y
3B
3A
3Y
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Figure eA.1 Common 74xx-series logic gates

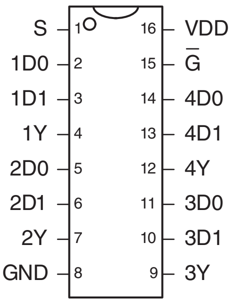

$\overline{1G}$
1
16
VDD
S1
2
15
$\overline{2G}$
1D3
3
14
S0
1D2
4
13
2D3
1D1
5
12
2D2
1D0
6
11
2D1
1Y
7
10
2D0
GND
8
9
2Y

74153 4:1 Mux

Two 4:1 Multiplexers

$D_{3:0}$:	data
$S_{1:0}$:	select
Y:	output
Gb:	enable

```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```

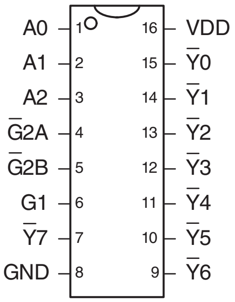

S
1
16
VDD
1D0
2
15
$\overline{G}$
1D1
3
14
4D0
1Y
4
13
4D1
2D0
5
12
4Y
2D1
6
11
3D0
2Y
7
10
3D1
GND
8
9
3Y

74157 2:1 Mux

Four 2:1 Multiplexers

$D_{1:0}$:	data
S:	select
Y:	output
Gb:	enable

```
always_comb
  if  (Gb) 1Y  = 0;
  else     1Y  = S ? 1D[1]  : 1D[0];
  if  (Gb) 2Y  = 0;
  else     2Y  = S ? 2D[1]  : 2D[0];
  if  (Gb) 3Y  = 0;
  else     3Y  = S ? 3D[1]  : 3D[0];
  if  (Gb) 4Y  = 0;
  else     4Y  = S ? 4D[1]  : 4D[0];
```

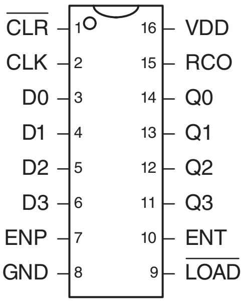

A0
1
16
VDD
A1
2
15
$\overline{Y0}$
A2
3
14
$\overline{Y1}$
$\overline{G2A}$
4
13
$\overline{Y2}$
$\overline{G2B}$
5
12
$\overline{Y3}$
G1
6
11
$\overline{Y4}$
$\overline{Y7}$
7
10
$\overline{Y5}$
GND
8
9
$\overline{Y6}$

74138 3:8 Decoder

3:8 Decoder

$A_{2:0}$:	address
$Yb_{7:0}$:	output
G1:	active high enable
G2:	active low enables

G1	$\overline{G2A}$	$\overline{G2B}$	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111


$\overline{CLR}$
1
16
VDD
CLK
2
15
RCO
D0
3
14
Q0
D1
4
13
Q1
D2
5
12
Q2
D3
6
11
Q3
ENP
7
10
ENT
GND
8
9
$\overline{LOAD}$

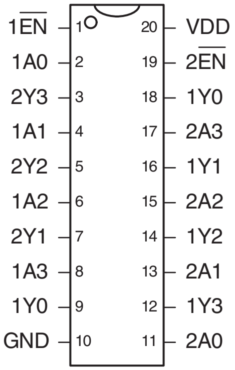
74161/163 Counter

4-bit Counter

CLK:	clock
$Q_{3:0}$:	counter output
$D_{3:0}$:	parallel input
CLRb:	async reset (161) sync reset (163)
LOADb:	load Q from D
ENP, ENT:	enables
RCO:	ripple carry out

```
always_ff @(posedge CLK)  // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```

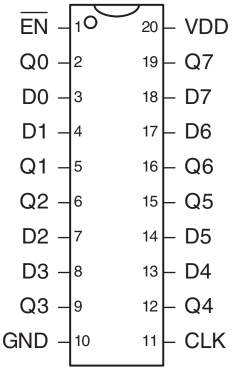

$\overline{1EN}$
1
20
VDD
1A0
2
19
$\overline{2EN}$
2Y3
3
18
1Y0
1A1
4
17
2A3
2Y2
5
16
1Y1
1A2
6
15
2A2
2Y1
7
14
1Y2
1A3
8
13
2A1
1Y0
9
12
1Y3
GND
10
11
2A0

74244 Tristate Buffer

8-bit Tristate Buffer

$A_{3:0}$:	input
$Y_{3:0}$:	output
ENb:	enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```


$\overline{EN}$
1
20
VDD
Q0
2
19
Q7
D0
3
18
D7
D1
4
17
D6
Q1
5
16
Q6
Q2
6
15
Q5
D2
7
14
D5
D3
8
13
D4
Q3
9
12
Q4
GND
10
11
CLK

74377 Register

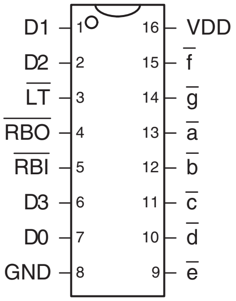
8-bit Enableable Register

CLK:	clock
$D_{7:0}$:	data
$Q_{7:0}$:	output
ENb:	enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Note: SystemVerilog variable names cannot start with numbers, but the names in the example code in Figure A.2 are chosen to match the manufacturer's data sheet.

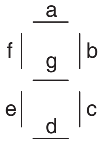
Figure eA.2 Medium-scale integration chips


D1
1
16
VDD
D2
2
15
$\overline{f}$
$\overline{LT}$
3
14
$\overline{g}$
$\overline{RBO}$
4
13
$\overline{a}$
$\overline{RBI}$
5
12
$\overline{b}$
D3
6
11
$\overline{c}$
D0
7
10
$\overline{d}$
GND
8
9
$\overline{e}$

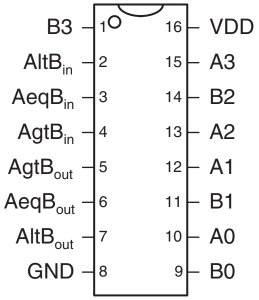
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f : segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



$\overline{RBO}$	$\overline{LT}$	$\overline{RBI}$	D3:0	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	$\overline{g}$
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

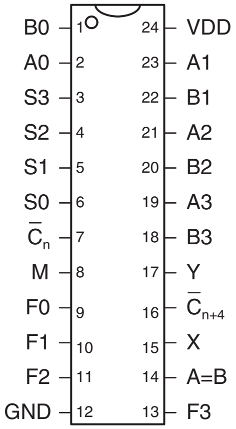

B3
1
16
VDD
$AltB_{in}$
2
15
A3
$AeqB_{in}$
3
14
B2
$AgtB_{in}$
4
13
A2
$AgtB_{out}$
5
12
A1
$AeqB_{out}$
6
11
B1
$AltB_{out}$
7
10
A0
GND
8
9
B0

7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always_comb
  if (A > B | (A == B & AgtBin)) begin
    AgtBout = 1; AeqBout = 0; AltBout = 0;
  end
  else if (A < B | (A == B & AltBin) begin
    AgtBout = 0; AeqBout = 0; AltBout = 1;
  end else begin
    AgtBout = 0; AeqBout = 1; AltBout = 0;
  end
```


B0
1
24
VDD
A0
2
23
A1
S3
3
22
B1
S2
4
21
A2
S1
5
20
B2
S0
6
19
A3
$\overline{C}_n$
7
18
B3
M
8
17
Y
F0
9
16
$\overline{C}_{n+4}$
F1
10
15
X
F2
11
14
A=B
GND
12
13
F3

74181 ALU

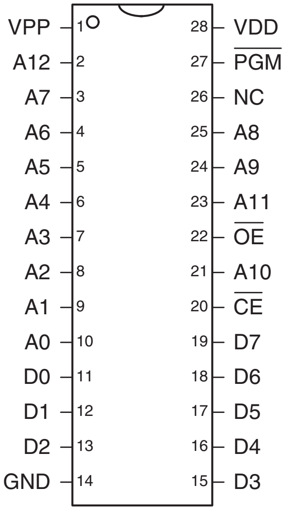
4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M : mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB : equality (in some modes)
X,Y : carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A        :  A        +  B       + ~Cbn;
```

```
always_comb
  case (F)
    0000: Y = M ? ~A        :  A                    + ~Cbn;
    0001: Y = M ? ~(A | B)  :  A       +  B         + ~Cbn;
    0010: Y = M ? (~A) & B  :  A       + ~B         + ~Cbn;
    0011: Y = M ? 4'b0000   :  4'b1111              + ~Cbn;
    0100: Y = M ? ~(A & B)  :  A       + (A & ~B)   + ~Cbn;
    0101: Y = M ? ~B        : (A | B)  + (A & ~B)   + ~Cbn;
    0110: Y = M ? A ^ B     :  A       - B          -  Cbn;
    0111: Y = M ? A & ~B    : (A & ~B)              -  Cbn;
    1000: Y = M ? ~A + B    :  A       + (A & B)    + ~Cbn;
    1001: Y = M ? ~(A ^ B)  :  A       +  B         + ~Cbn;
    1010: Y = M ? B         : (A | ~B) + (A & B)    + ~Cbn;
    1011: Y = M ? A & B     : (A & B)               + ~Cbn;
    1100: Y = M ? 1         :  A       +  A         + ~Cbn;
    1101: Y = M ? A | ~B    : (A | B)  +  A         + ~Cbn;
    1110: Y = M ? A | B     : (A | ~B) +  A         + ~Cbn;
    1111: Y = M ? A         :  A                    -  Cbn;
  endcase
```

Figure eA.3 More medium-scale integration (MSI) chips



8 KB EPROM

$A_{12:0}$:	address input
$D_{7:0}$:	data output
CEb :	chip enable
OEb :	output enable
PGMb :	program
VPP :	program voltage
NC :	no connection

```
assign D = (~CEb & ~OEb) ? ROM[A]
                         : 8'bz;
```

Figure eA.4 **2764 8KB EPROM**

replacing the contents of the PROM rather than rewiring connections between chips. Lookup tables are useful for small functions but become prohibitively expensive as the number of inputs grows.

For example, the classic 2764 8-KB (64-Kb) erasable PROM (EPROM) is shown in Figure eA.4. The EPROM has 13 address lines to specify one of the 8K words and 8 data lines to read the byte of data at that word. The chip enable and output enable must both be asserted for data to be read. The maximum propagation delay is 200 ps. In normal operation, $\overline{PGM} = 1$ and VPP is not used. The EPROM is usually programmed on a special programmer that sets $\overline{PGM} = 0$, applies 13 V to VPP, and uses a special sequence of inputs to configure the memory.

Modern PROMs are similar in concept but have much larger capacities and more pins. Flash memory is the cheapest type of PROM, selling for about $0.30 per gigabyte in 2015. Prices have historically declined by 30 to 40% per year.

A.3.2 PLAs

As discussed in Section 5.6.1, PLAs contain AND and OR planes to compute any combinational function written in sum-of-products form. The AND and OR planes can be programmed using the same techniques for PROMs. A PLA has two columns for each input and one column for each output. It has one row for each minterm. This organization is more efficient than a PROM for many functions, but the array still grows excessively large for functions with numerous I/Os and minterms.

Many different manufacturers have extended the basic PLA concept to build *programmable logic devices* (*PLDs*) that include registers. The 22V10 is one of the most popular classic PLDs. It has 12 dedicated input pins and 10 outputs. The outputs can come directly from the PLA or from clocked registers on the chip. The outputs can also be fed back into the PLA. Thus, the 22V10 can directly implement FSMs with up to 12 inputs, 10 outputs, and 10 bits of state. The 22V10 costs about \$2 in quantities of 100. PLDs have been rendered mostly obsolete by the rapid improvements in capacity and cost of FPGAs.

A.3.3 FPGAs

As discussed in Section 5.6.2, FPGAs consist of arrays of configurable *logic elements* (*LEs*), also called *configurable logic blocks* (*CLBs*), connected together with programmable wires. The LEs contain small lookup tables and flip-flops. FPGAs scale gracefully to extremely large capacities, with thousands of lookup tables. Xilinx and Altera are two of the leading FPGA manufacturers.

Lookup tables and programmable wires are flexible enough to implement any logic function. However, they are an order of magnitude less efficient in speed and cost (chip area) than hard-wired versions of the same functions. Thus, FPGAs often include specialized blocks, such as memories, multipliers, and even entire microprocessors.

Figure eA.5 shows the design process for a digital system on an FPGA. The design is usually specified with a hardware description language (HDL), although some FPGA tools also support schematics. The design is then simulated. Inputs are applied and compared against expected outputs to *verify* that the logic is correct. Usually some debugging is required. Next, logic *synthesis* converts the HDL into Boolean functions. Good synthesis tools produce a schematic of the functions, and the prudent designer examines these schematics, as well as any warnings produced during synthesis, to ensure that the desired logic was produced. Sometimes sloppy coding leads to circuits that are much larger than intended or to circuits with asynchronous logic. When the synthesis results are good, the FPGA tool *maps* the functions onto the LEs of a specific chip. The *place and route* tool determines which functions go in which lookup tables and how they are wired together. Wire delay increases with length, so critical circuits should be placed close together. If the design is too big to fit on the chip, it must be reengineered. *Timing analysis* compares the timing constraints (e.g., an intended clock speed of 100 MHz) against the actual circuit delays and reports any errors. If the logic is too slow, it may have to be redesigned or pipelined differently. When the design is correct, a file is generated specifying the contents of all the LEs and the programming of all the wires on the FPGA. Many FPGAs store this *configuration* information in static RAM that must be reloaded each time the FPGA is

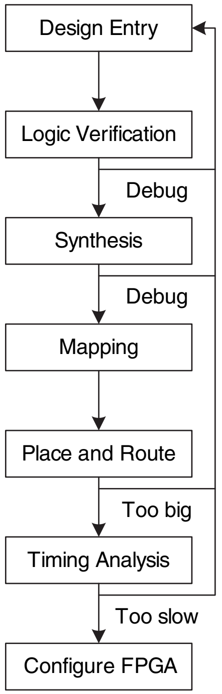


Figure eA.5 **FPGA design flow**

turned on. The FPGA can download this information from a computer in the laboratory, or can read it from a nonvolatile ROM when power is first applied.

Example eA.1 FPGA TIMING ANALYSIS

Alyssa P. Hacker is using an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. Her design is implemented as an FSM, and she is using a Cyclone IV FPGA. According to the data sheet, the FPGA has the timing characteristics shown in Table eA.1.

Alyssa would like her FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which her FSM could possibly run?

Solution: At 100 MHz, the cycle time, T_c, is 10 ns. Alyssa uses Equation 3.14 to figure out the minimum combinational propagation delay, t_{pd}, at this cycle time:

$$t_{pd} \leq 10\,\text{ns} - (0.199\,\text{ns} + 0.076\,\text{ns}) = 9.725\,\text{ns} \qquad \text{(A.1)}$$

With a combined LE and wire delay of 381 ps + 246 ps = 627 ps, Alyssa's FSM can use at most 15 consecutive LEs (9.725/0.627) to implement the next-state logic.

The fastest speed at which an FSM will run on this Cyclone IV FPGA is when it is using a single LE for the next state logic. The minimum cycle time is

$$T_c \geq 381\,\text{ps} + 199\,\text{ps} + 76\,\text{ps} = 656\,\text{ps} \qquad \text{(A.2)}$$

Therefore, the maximum frequency is 1.5 GHz.

Table eA.1 **Cyclone IV timing**

Name	Value (ps)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Altera advertises the Cyclone IV FPGA with 14,400 LEs for $25 in 2015. In large quantities, medium-sized FPGAs typically cost several dollars. The largest FPGAs cost hundreds or even thousands of dollars.

The cost has declined at approximately 30% per year, so FPGAs are becoming extremely popular.

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cell phone chips are common examples of ASICs. The ASIC designer places transistors to form logic gates and wires the gates together. Because the ASIC is hardwired for a specific function, it is typically several times faster than an FPGA and occupies an order of magnitude less chip area (and hence cost) than an FPGA with the same function. However, the *masks* specifying where transistors and wires are located on the chip cost hundreds of thousands of dollars to produce. The fabrication process usually requires 6 to 12 weeks to manufacture, package, and test the ASICs. If errors are discovered after the ASIC is manufactured, the designer must correct the problem, generate new masks, and wait for another batch of chips to be fabricated. Hence, ASICs are suitable only for products that will be produced in large quantities and whose function is well defined in advance.

Figure eA.6 shows the ASIC design process, which is similar to the FPGA design process of Figure eA.5. Logic verification is especially important because correction of errors after the masks are produced is expensive. Synthesis produces a *netlist* consisting of logic gates and connections between the gates; the gates in this netlist are placed, and the wires are routed between gates. When the design is satisfactory, masks are generated and used to fabricate the ASIC. A single speck of dust can ruin an ASIC, so the chips must be tested after fabrication. The fraction of manufactured chips that work is called the *yield*; it is typically 50 to 90%, depending on the size of the chip and the maturity of the manufacturing process. Finally, the working chips are placed in packages, as will be discussed in Section A.7.

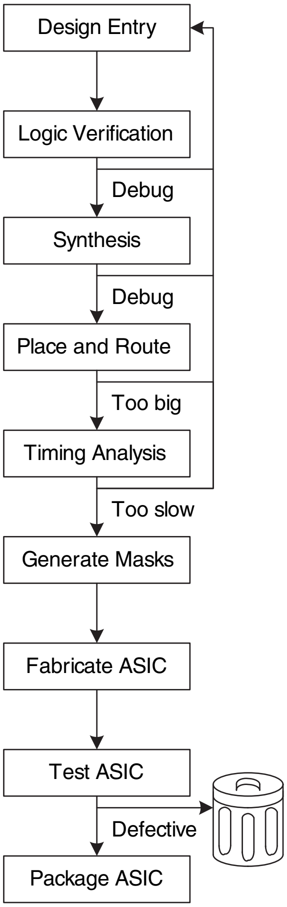


Figure eA.6 **ASIC design flow**

A.5 DATA SHEETS

Integrated circuit manufacturers publish *data sheets* that describe the functions and performance of their chips. It is essential to read and understand the data sheets. One of the leading sources of errors in digital systems comes from misunderstanding the operation of a chip.

Data sheets are usually available from the manufacturer's Web site. If you cannot locate the data sheet for a part and do not have clear documentation from another source, don't use the part. Some of the entries in the data sheet may be cryptic. Often the manufacturer publishes data books containing data sheets for many related parts. The beginning of the data book has additional explanatory information. This information can usually be found on the Web with a careful search.

This section dissects the Texas Instruments (TI) data sheet for a 74HC04 inverter chip. The data sheet is relatively simple but illustrates many of the major elements. TI still manufacturers a wide variety of 74xx-series chips. In the past, many other companies built these chips too, but the market is consolidating as the sales decline.

Figure eA.7 shows the first page of the data sheet. Some of the key sections are highlighted in blue. The title is SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS means that the chip contains six inverters. SN indicates that TI is the manufacturer. Other manufacture codes include MC for Motorola and DM for National Semiconductor. You can generally ignore these codes, because all of the manufacturers build compatible 74xx-series logic. HC is the logic family (high speed CMOS). The logic family determines the speed and power consumption of the chip, but not the function. For example, the 7404, 74HC04, and 74LS04 chips all contain six inverters, but they differ in performance and cost. Other logic families are discussed in Section A.6. The 74xx chips operate across the commercial or industrial temperature range (0 to 70 °C or −40 to 85 °C, respectively), whereas the 54xx chips operate across the military temperature range (−55 to 125 °C) and sell for a higher price but are otherwise compatible.

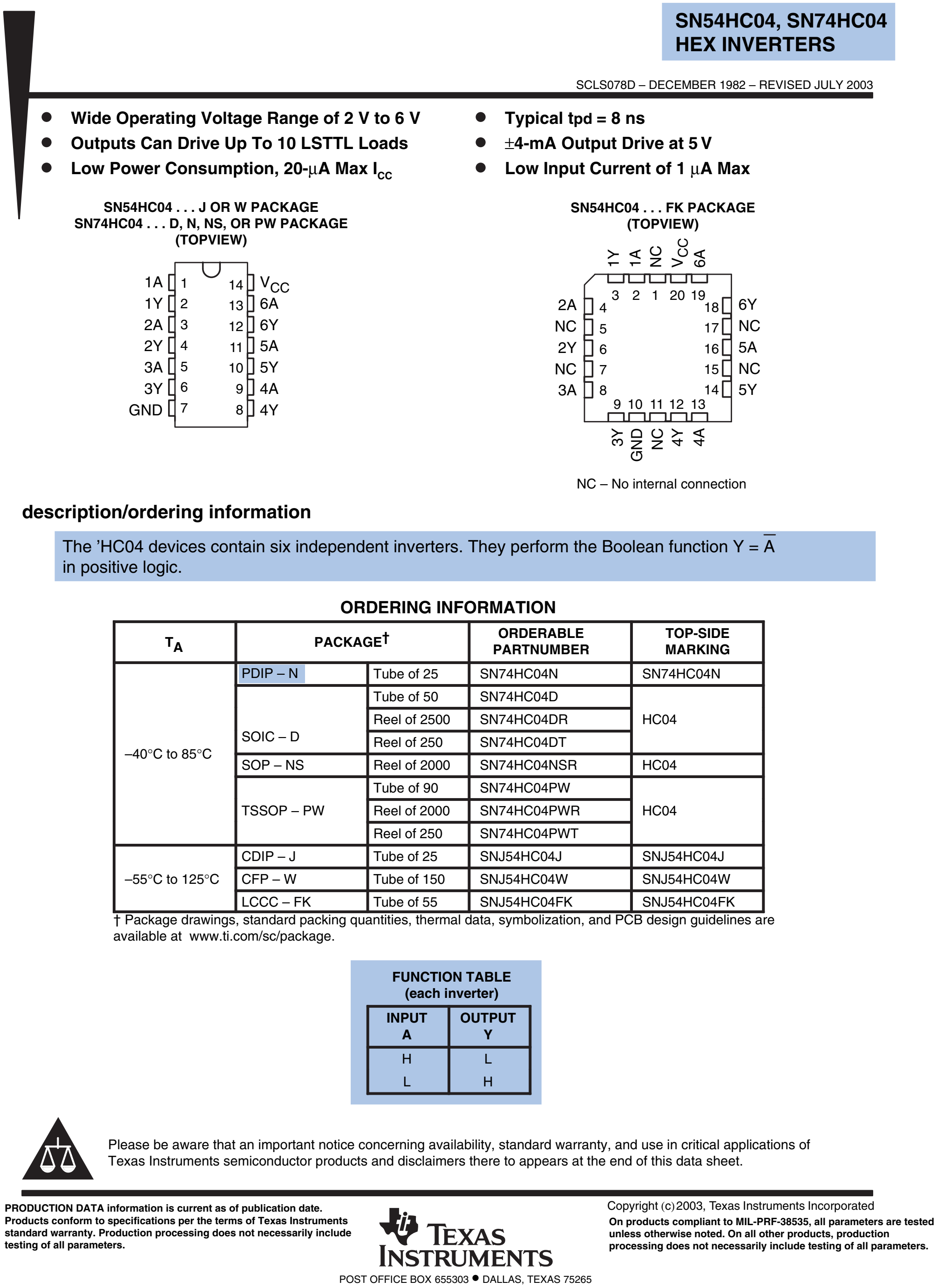
The 7404 is available in many different packages, and it is important to order the one you intended when you make a purchase. The packages are distinguished by a suffix on the part number. N indicates a *plastic dual inline package* (*PDIP*), which fits in a breadboard or can be soldered in through-holes in a printed circuit board. Other packages are discussed in Section A.7.

The function table shows that each gate inverts its input. If A is HIGH (H), Y is LOW (L) and vice versa. The table is trivial in this case but is more interesting for more complex chips.

Figure eA.8 shows the second page of the data sheet. The logic diagram indicates that the chip contains inverters. The *absolute maximum* section indicates conditions beyond which the chip could be destroyed. In particular, the power supply voltage (V_{CC}, also called V_{DD} in this book) should not exceed 7 V. The continuous output current should not exceed 25 mA. The *thermal resistance* or impedance, θ_{JA}, is used to calculate the temperature rise caused by the chip's dissipating power. If the *ambient* temperature in the vicinity of the chip is T_A and the chip dissipates P_{chip}, then the temperature on the chip itself at its *junction* with the package is

$$T_J = T_A + P_{\text{chip}}\,\theta_{JA} \tag{A.3}$$

For example, if a 7404 chip in a plastic DIP package is operating in a hot box at 50 °C and consumes 20 mW, the junction temperature will climb



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- **Wide Operating Voltage Range of 2 V to 6 V**
- **Outputs Can Drive Up To 10 LSTTL Loads**
- **Low Power Consumption, 20-µA Max I_{CC}**
- **Typical tpd = 8 ns**
- **±4-mA Output Drive at 5 V**
- **Low Input Current of 1 µA Max**

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)

Pin	Name	Pin	Name
1	1A	14	V_{CC}
2	1Y	13	6A
3	2A	12	6Y
4	2Y	11	5A
5	3A	10	5Y
6	3Y	9	4A
7	GND	8	4Y

SN54HC04 . . . FK PACKAGE
(TOPVIEW)

Pin	Name	Pin	Name	Pin	Name	Pin	Name
3	1Y	4	2A	18	6Y	9	3Y
2	1A	5	NC	17	NC	10	GND
1	NC	6	2Y	16	5A	11	NC
20	V_{CC}	7	NC	15	NC	12	4Y
19	6A	8	3A	14	5Y	13	4A

NC – No internal connection

description/ordering information

The ’HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this data sheet.

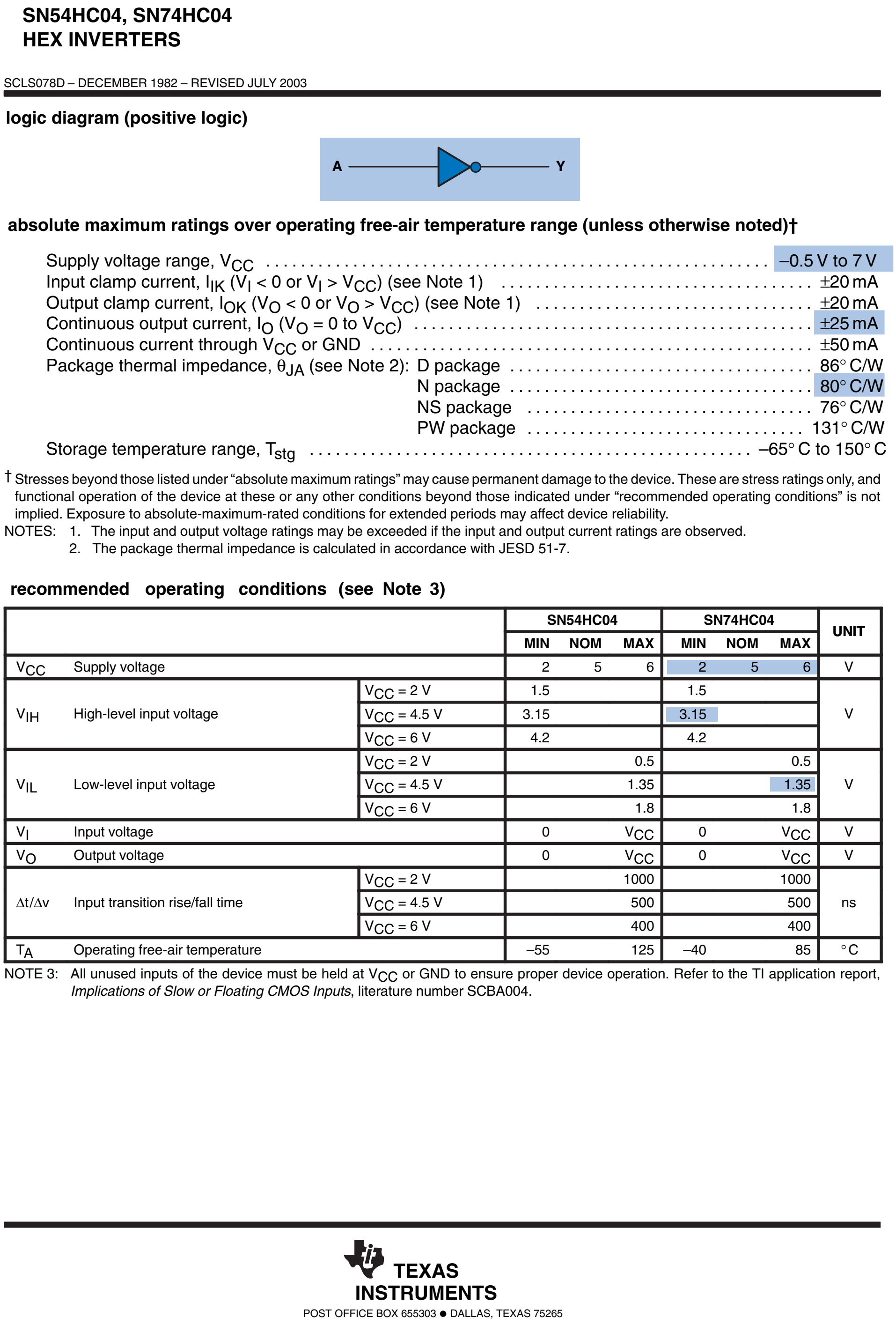
PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

Copyright (c) 2003, Texas Instruments Incorporated

On products compliant to MIL-PRF-38535, all parameters are tested unless otherwise noted. On all other products, production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Figure eA.7 **7404 data sheet page 1**



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC}		–0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1)		±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1)		±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC})		±25 mA
Continuous current through V_{CC} or GND		±50 mA
Package thermal impedance, θ_{JA} (see Note 2):	D package	86° C/W
	N package	80° C/W
	NS package	76° C/W
	PW package	131° C/W
Storage temperature range, T_{stg}		–65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 ● DALLAS, TEXAS 75265

Figure eA.8 7404 data sheet page 2

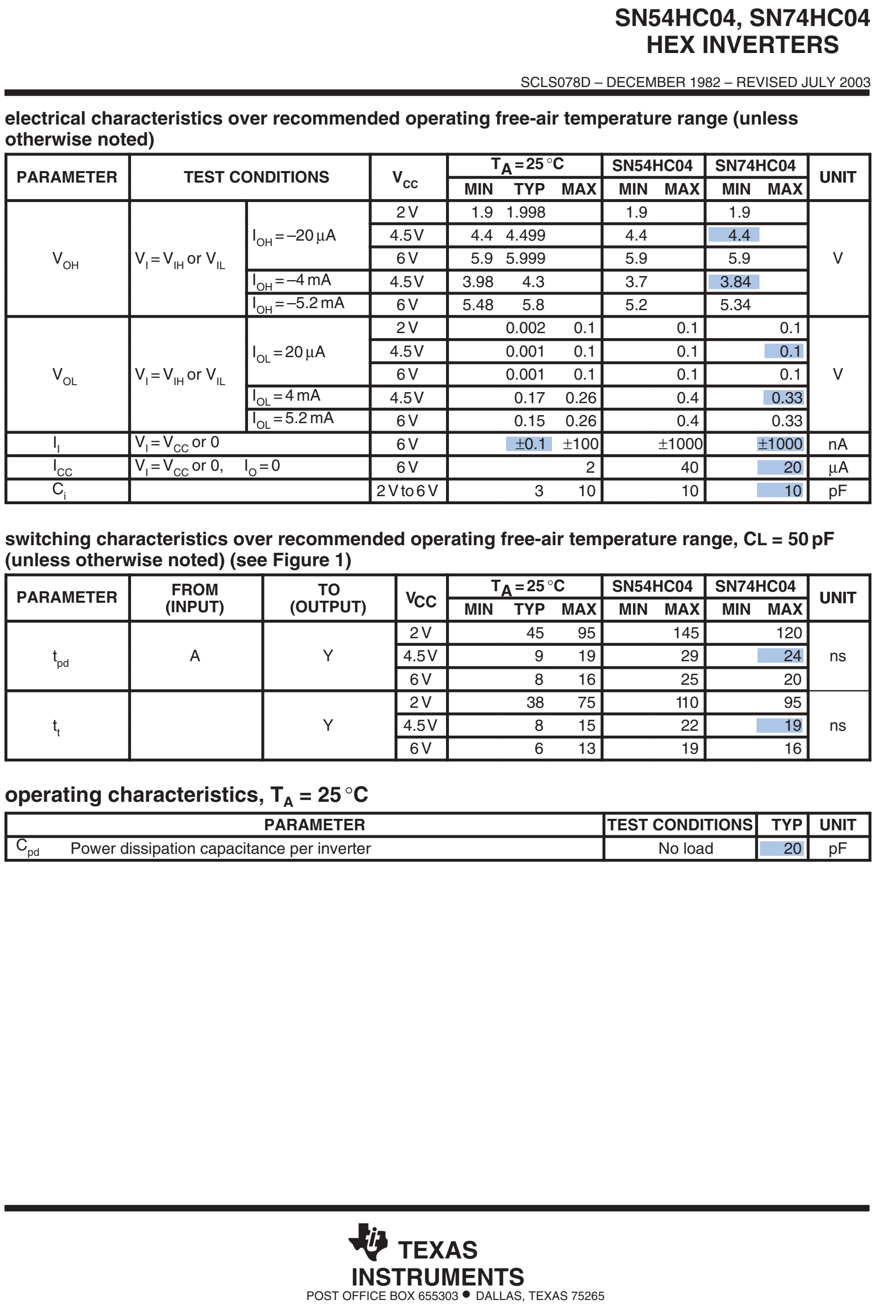
to 50 °C + 0.02 W × 80 °C/W = 51.6 °C. Internal power dissipation is seldom important for 74xx-series chips, but it becomes important for modern chips that dissipate tens of watts or more.

The *recommended operating conditions* define the environment in which the chip should be used. Within these conditions, the chip should meet specifications. These conditions are more stringent than the absolute maximums. For example, the power supply voltage should be between 2 and 6 V. The input logic levels for the HC logic family depend on V_{DD}. Use the 4.5 V entries when $V_{DD} = 5$ V, to allow for a 10% droop in the power supply caused by noise in the system.

Figure eA.9 shows the third page of the data sheet. The *electrical characteristics* describe how the device performs when used within the recommended operating conditions if the inputs are held constant. For example, if $\mathrm{V_{CC}} = 5$ V (and droops to 4.5 V) and the output current I_{OH}/I_{OL} does not exceed 20 μ*A*, $V_{OH} = 4.4$ V and $V_{OL} = 0.1$ V in the worst case. If the output current increases, the output voltages become less ideal, because the transistors on the chip struggle to provide the current. The HC logic family uses CMOS transistors that draw very little current. The current into each input is guaranteed to be less than 1000 nA and is typically only 0.1 nA at room temperature. The *quiescent* power supply current (I_{DD}) drawn while the chip is idle is less than 20 μ*A*. Each input has less than 10 pF of capacitance.

The *switching characteristics* define how the device performs when used within the recommended operating conditions if the inputs change. The *propagation delay*, t_{pd}, is measured from when the input passes through 0.5 V_{CC} to when the output passes through 0.5 V_{CC}. If V_{CC} is nominally 5 V and the chip drives a capacitance of less than 50 pF, the propagation delay will not exceed 24 ns (and typically will be much faster). Recall that each input may present 10 pF, so the chip cannot drive more than five identical chips at full speed. Indeed, stray capacitance from the wires connecting chips cuts further into the useful load. The *transition time*, also called the rise/fall time, is measured as the output transitions between 0.1 V_{CC} and 0.9 V_{CC}.

Recall from Section 1.8 that chips consume both *static* and *dynamic power*. Static power is low for HC circuits. At 85 °C, the maximum quiescent supply current is 20 μ*A*. At 5 V, this gives a static power consumption of 0.1 mW. The dynamic power depends on the capacitance being driven and the switching frequency. The 7404 has an internal power dissipation capacitance of 20 pF per inverter. If all six inverters on the 7404 switch at 10 MHz and drive external loads of 25 pF, then the dynamic power given by Equation 1.4 is $\frac{1}{2}(6)(20\text{ pF} + 25\text{ pF})(5^2)(10\text{ MHz}) = 33.75\text{ mW}$ and the maximum total power is 33.85 mW.



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	$T_A = 25\,°C$			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	$I_{OH} = -20\,\mu A$	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		$I_{OH} = -4$ mA	4.5 V	3.98	4.3		3.7		3.84		
		$I_{OH} = -5.2$ mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	$I_{OL} = 20\,\mu A$	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		$I_{OL} = 4$ mA	4.5 V		0.17	0.26		0.4		0.33	
		$I_{OL} = 5.2$ mA	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, $I_O = 0$		6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	VCC	$T_A = 25\,°C$			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, $T_A = 25\,°C$

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 ● DALLAS, TEXAS 75265

Figure eA.9 **7404 data sheet page 3**

A.6 LOGIC FAMILIES

The 74xx-series logic chips have been manufactured using many different technologies, called *logic families*, that offer different speed, power, and logic level trade-offs. Other chips are usually designed to be compatible with some of these logic families. The original chips, such as the 7404, were built using bipolar transistors in a technology called *Transistor-Transistor Logic* (*TTL*). Newer technologies add one or more letters after the 74 to indicate the logic family, such as 74LS04, 74HC04, or 74AHCT04. Table eA.2 summarizes the most common 5-V logic families.

Advances in bipolar circuits and process technology led to the *Schottky* (*S*) and *Low-Power Schottky* (*LS*) families. Both are faster than TTL. Schottky draws more power, whereas Low-Power Schottky draws less. *Advanced Schottky* (*AS*) and *Advanced Low-Power Schottky* (*ALS*) have improved speed and power compared to S and LS. *Fast* (*F*) logic is faster and draws less power than AS. All of these families provide more current for LOW outputs than for HIGH outputs and hence have

Table eA.2 **Typical specifications for 5-V logic families**

	Bipolar / TTL						CMOS		CMOS / TTL Compatible	
Characteristic	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH} (V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL} (V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH} (V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL} (V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH} (mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL} (mA)	16	20	8	20	8	20	4	8	4	8
I_{IL} (mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH} (mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD} (mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{pd} (pF)	n/a						20	12	20	14
cost* (US $)	obsolete	0.63	0.25	0.53	0.32	0.22	0.12	0.12	0.12	0.12

*Per unit in quantities of 1000 for the 7408 from Texas Instruments in 2012.

asymmetric logic levels. They conform to the "TTL" logic levels: $V_{IH} = 2$ V, $V_{IL} = 0.8$ V, $V_{OH} > 2.4$ V, and $V_{OL} < 0.5$ V.

As CMOS circuits matured in the 1980s and 1990s, they became popular because they draw very little power supply or input current. The *High Speed CMOS* (*HC*) and *Advanced High Speed CMOS* (*AHC*) families draw almost no static power. They also deliver the same current for HIGH and LOW outputs. They conform to the "CMOS" logic levels: $V_{IH} = 3.15$ V, $V_{IL} = 1.35$ V, $V_{OH} > 3.8$ V, and $V_{OL} < 0.44$ V. Unfortunately, these levels are incompatible with TTL circuits, because a TTL HIGH output of 2.4 V may not be recognized as a legal CMOS HIGH input. This motivates the use of *High Speed TTL-compatible CMOS* (*HCT*) and *Advanced High Speed TTL-compatible CMOS* (*AHCT*), which accept TTL input logic levels and generate valid CMOS output logic levels. These families are slightly slower than their pure CMOS counterparts. All CMOS chips are sensitive to *electrostatic discharge* (*ESD*) caused by static electricity. Ground yourself by touching a large metal object before handling CMOS chips, lest you zap them.

The 74xx-series logic is inexpensive. The newer logic families are often cheaper than the obsolete ones. The LS family is widely available and robust and is a popular choice for laboratory or hobby projects that have no special performance requirements.

The 5-V standard collapsed in the mid-1990s, when transistors became too small to withstand the voltage. Moreover, lower voltage offers lower power consumption. Now 3.3, 2.5, 1.8, 1.2, and even lower voltages are commonly used. The plethora of voltages raises challenges in communicating between chips with different power supplies. Table eA.3 lists some of the low-voltage logic families. Not all 74xx parts are available in all of these logic families.

All of the low-voltage logic families use CMOS transistors, the workhorse of modern integrated circuits. They operate over a wide range of V_{DD}, but the speed degrades at lower voltage. *Low-Voltage CMOS* (*LVC*) logic and *Advanced Low-Voltage CMOS* (*ALVC*) logic are commonly used at 3.3, 2.5, or 1.8 V. LVC withstands inputs up to 5.5 V, so it can receive inputs from 5-V CMOS or TTL circuits. *Advanced Ultra-Low-Voltage CMOS* (*AUC*) is commonly used at 2.5, 1.8, or 1.2 V and is exceptionally fast. Both ALVC and AUC withstand inputs up to 3.6 V, so they can receive inputs from 3.3-V circuits.

FPGAs often offer separate voltage supplies for the internal logic, called the *core*, and for the input/output (I/O) pins. As FPGAs have advanced, the core voltage has dropped from 5 to 3.3, 2.5, 1.8, and 1.2 V to save power and avoid damaging the very small transistors. FPGAs have configurable I/Os that can operate at many different voltages, so as to be compatible with the rest of the system.

Table eA.3 **Typical specifications for low-voltage logic families**

	LVC			ALVC			AUC		
V_{dd} (V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd} (ns)	4.1	6.9	9.8	2.8	3	?[*]	1.8	2.3	3.4
V_{IH} (V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL} (V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH} (V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL} (V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O (mA)	24	8	4	24	12	12	9	8	3
I_I (mA)	0.02			0.005			0.005		
I_{DD} (mA)	0.01			0.01			0.01		
C_{pd} (pF)	10	9.8	7	27.5	23	?[*]	17	14	14
cost (US $)	0.17			0.20			not available		

[*] Delay and capacitance not available at the time of writing.

A.7 PACKAGING AND ASSEMBLY

Integrated circuits are typically placed in *packages* made of plastic or ceramic. The packages serve a number of functions, including connecting the tiny metal I/O pads of the chip to larger pins in the package for ease of connection, protecting the chip from physical damage, and spreading the heat generated by the chip over a larger area to help with cooling. The packages are placed on a breadboard or printed circuit board and wired together to assemble the system.

Packages

Figure eA.10 shows a variety of integrated circuit packages. Packages can be generally categorized as *through-hole* or *surface mount* (*SMT*). Through-hole packages, as their name implies, have pins that can be inserted through holes in a printed circuit board or into a socket. *Dual inline packages* (*DIPs*) have two rows of pins with 0.1-inch spacing between pins. *Pin grid arrays* (*PGAs*) support more pins in a smaller package by placing the pins under the package. SMT packages are soldered directly to the surface of a printed circuit board without using holes. Pins on SMT parts are called *leads*. The *thin small outline package* (*TSOP*) has two rows of closely spaced leads (typically 0.02-inch spacing). *Plastic leaded chip carriers* (*PLCCs*) have J-shaped leads on all four

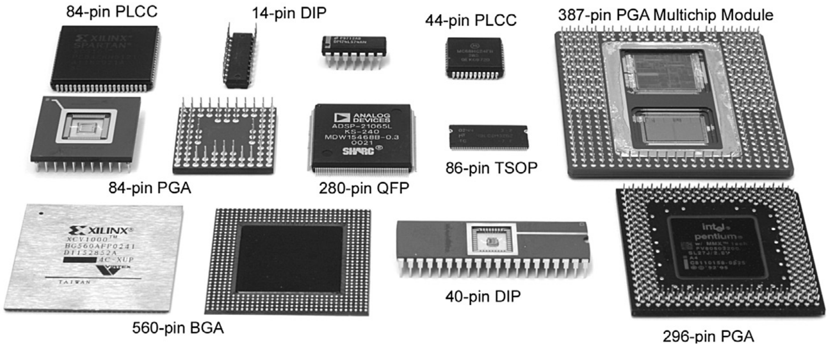


Figure eA.10 **Integrated circuit packages**

sides, with 0.05-inch spacing. They can be soldered directly to a board or placed in special sockets. *Quad flat packs* (*QFPs*) accommodate a large number of pins using closely spaced legs on all four sides. *Ball grid arrays* (*BGAs*) eliminate the legs altogether. Instead, they have hundreds of tiny solder balls on the underside of the package. They are carefully placed over matching pads on a printed circuit board, then heated so that the solder melts and joins the package to the underlying board.

Breadboards

DIPs are easy to use for prototyping, because they can be placed in a *breadboard*. A breadboard is a plastic board containing rows of sockets, as shown in Figure eA.11. All five holes in a row are connected together. Each pin of the package is placed in a hole in a separate row. Wires can be placed in adjacent holes in the same row to make connections to the pin. Breadboards often provide separate columns of connected holes running the height of the board to distribute power and ground.

Figure eA.11 shows a breadboard containing a majority gate built with a 74LS08 AND chip and a 74LS32 OR chip. The schematic of the circuit is shown in Figure eA.12. Each gate in the schematic is labeled with the chip (08 or 32) and the pin numbers of the inputs and outputs (see Figure eA.1). Observe that the same connections are made on the breadboard. The inputs are connected to pins 1, 2, and 5 of the 08 chip, and the output is measured at pin 6 of the 32 chip. Power and ground are connected to pins 14 and 7, respectively, of each chip, from the vertical power and ground columns that are attached to the banana plug receptacles, Vb and Va. Labeling the schematic in this way and checking off connections as they are made is a good way to reduce the number of mistakes made during breadboarding.

Unfortunately, it is easy to accidentally plug a wire in the wrong hole or have a wire fall out, so breadboarding requires a great deal of care (and usually some debugging in the laboratory). Breadboards are suited only to prototyping, not production.

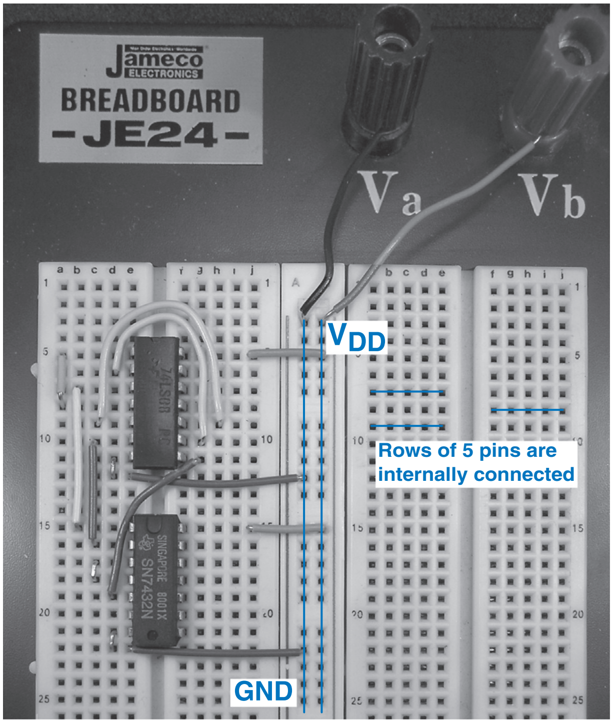

Jameco ELECTRONICS
BREADBOARD
-JE24-
Va
Vb
VDD
Rows of 5 pins are internally connected
GND
74LS08
SINGAPORE 8001X
SN7432N

Figure eA.11 Majority circuit on breadboard

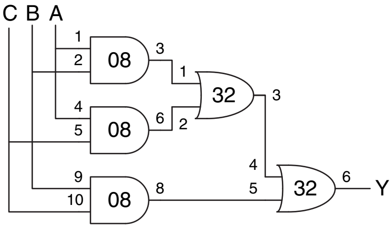

C B A
1
2
08
3
1
32
3
4
5
08
6
2
9
10
08
8
4
5
32
6
Y

Figure eA.12 Majority gate schematic with chips and pins identified

Printed Circuit Boards

Instead of breadboarding, chip packages may be soldered to a *printed circuit board* (*PCB*). The PCB is formed of alternating layers of conducting copper and insulating epoxy. The copper is etched to form wires called *traces*. Holes called *vias* are drilled through the board and plated with metal to connect between layers. PCBs are usually designed with *computer-aided design* (*CAD*) tools. You can etch and drill your own simple boards in the laboratory, or you can send the board design to a specialized factory for inexpensive mass production. Factories have turnaround times of days (or weeks, for cheap mass production runs) and typically charge a few hundred dollars in setup fees and a few dollars per board for moderately complex boards built in large quantities.

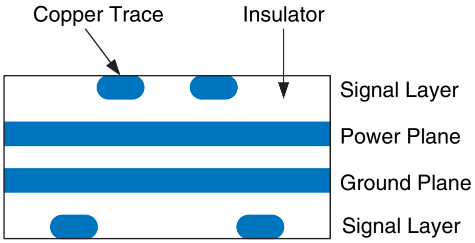


Figure eA.13 **Printed circuit board cross-section**

PCB traces are normally made of copper because of its low resistance. The traces are embedded in an insulating material, usually a green, fire-resistant plastic called FR4. A PCB also typically has copper power and ground layers, called *planes*, between signal layers. Figure eA.13 shows a cross-section of a PCB. The signal layers are on the top and bottom, and the power and ground planes are embedded in the center of the board. The power and ground planes have low resistance, so they distribute stable power to components on the board. They also make the capacitance and inductance of the traces uniform and predictable.

Figure eA.14 shows a PCB for a 1970s vintage Apple II+ computer. At the top is a 6502 microprocessor. Beneath are six 16-Kb ROM chips forming 12 KB of ROM containing the operating system. Three rows of eight 16-Kb DRAM chips provide 48 KB of RAM. On the right are several rows of 74xx-series logic for memory address decoding and other functions. The lines between chips are traces that wire the chips together. The dots at the ends of some of the traces are vias filled with metal.

Putting It All Together

Most modern chips with large numbers of inputs and outputs use SMT packages, especially QFPs and BGAs. These packages require a printed circuit board rather than a breadboard. Working with BGAs is especially challenging because they require specialized assembly equipment. Moreover, the balls cannot be probed with a voltmeter or oscilloscope during debugging in the laboratory, because they are hidden under the package.

In summary, the designer needs to consider packaging early on to determine whether a breadboard can be used during prototyping and whether BGA parts will be required. Professional engineers rarely use breadboards when they are confident of connecting chips together correctly without experimentation.

A.8 TRANSMISSION LINES

We have assumed so far that wires are *equipotential* connections that have a single voltage along their entire length. Signals actually propagate along wires at the speed of light in the form of electromagnetic waves. If the wires are short enough or the signals change slowly, the equipotential

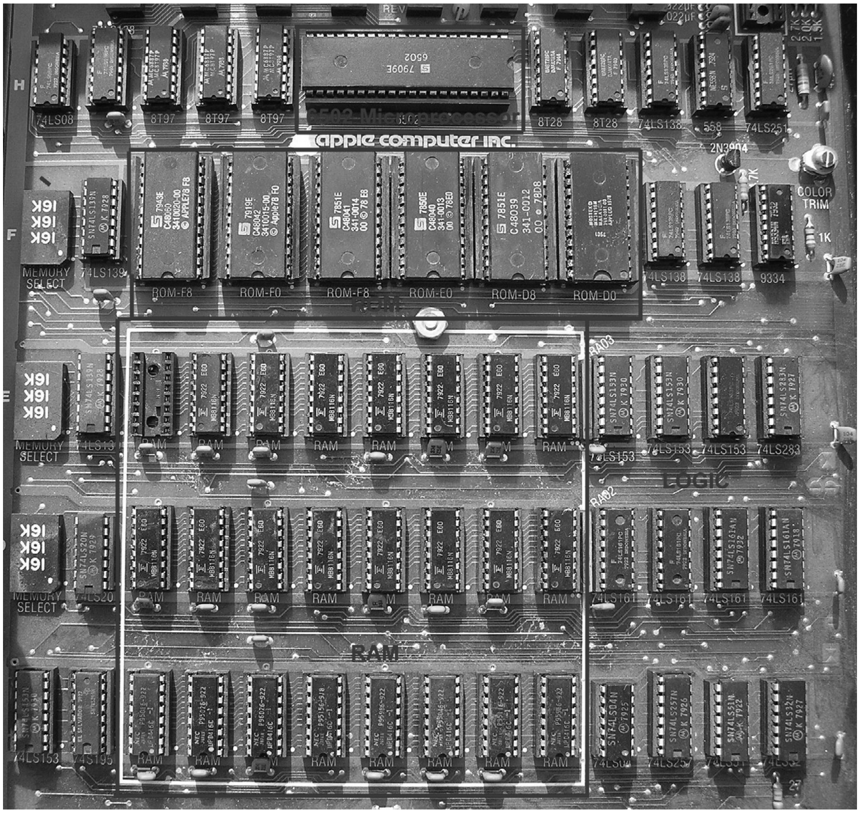


Figure eA.14 **Apple II+ circuit board**

assumption is good enough. When the wire is long or the signal is very fast, the *transmission time* along the wire becomes important to accurately determine the circuit delay. We must model such wires as *transmission lines*, in which a wave of voltage and current propagates at the speed of light. When the wave reaches the end of the line, it may reflect back along the line. The reflection may cause noise and odd behaviors unless steps are taken to limit it. Hence, the digital designer must consider transmission line behavior to accurately account for the delay and noise effects in long wires.

Electromagnetic waves travel at the speed of light in a given medium, which is fast but not instantaneous. The speed of light, ν, depends on the *permittivity*, ε, and *permeability*, μ, of the medium[1]: $\nu = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$.

---

[1] The capacitance, C, and inductance, L, of a wire are related to the permittivity and permeability of the physical medium in which the wire is located.

The speed of light in free space is $v = c = 3 \times 10^8$ m/s. Signals in a PCB travel at about half this speed, because the FR4 insulator has four times the permittivity of air. Thus, PCB signals travel at about 1.5×10^8 m/s, or 15 cm/ns. The time delay for a signal to travel along a transmission line of length l is

$$t_d = \frac{l}{v} \tag{A.4}$$

The *characteristic impedance* of a transmission line, Z_0 (pronounced "Z-naught"), is the ratio of voltage to current in a wave traveling along the line: $Z_0 = V/I$. It is *not* the resistance of the wire (a good transmission line in a digital system typically has negligible resistance). Z_0 depends on the inductance and capacitance of the line (see the derivation in Section A.8.7) and typically has a value of 50 to 75 Ω.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.5}$$

Figure eA.15 shows the symbol for a transmission line. The symbol resembles a *coaxial cable* with an inner signal conductor and an outer grounded conductor like that used in television cable wiring.

The key to understanding the behavior of transmission lines is to visualize the wave of voltage propagating along the line at the speed of light. When the wave reaches the end of the line, it may be absorbed or reflected, depending on the termination or load at the end. Reflections travel back along the line, adding to the voltage already on the line. Terminations are classified as matched, open, short, or mismatched. The following sections explore how a wave propagates along the line and what happens to the wave when it reaches the termination.

A.8.1 Matched Termination

Figure eA.16 shows a transmission line of length l with a *matched termination*, which means that the load impedance, Z_L, is equal to the characteristic impedance, Z_0. The transmission line has a characteristic impedance of 50 Ω. One end of the line is connected to a voltage source

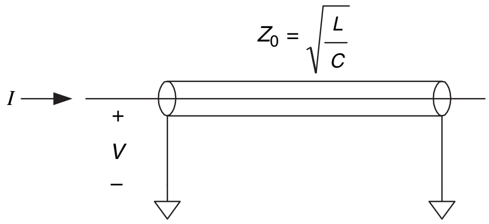


Figure eA.15 **Transmission line symbol**

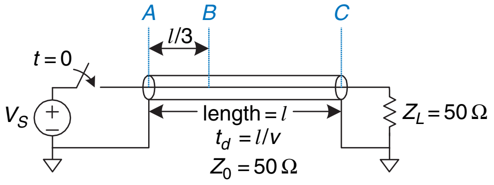


Figure eA.16 Transmission line with matched termination

through a switch that closes at time $t = 0$. The other end is connected to the 50 Ω matched load. This section analyzes the voltages and currents at points A, B, and C—at the beginning of the line, one-third of the length along the line, and at the end of the line, respectively.

Figure eA.17 shows the voltages at points A, B, and C over time. Initially, there is no voltage or current flowing in the transmission line, because the switch is open. At time $t = 0$, the switch closes, and the voltage source launches a wave with voltage $V = V_S$ along the line. This is called the *incident wave*. Because the characteristic impedance is Z_0, the wave has current $I = V_S/Z_0$. The voltage reaches the beginning of the line (point A) immediately, as shown in Figure eA.17(a). The wave propagates along the line at the speed of light. At time $t_d/3$, the wave reaches point B. The voltage at this point abruptly rises from 0 to V_S, as shown in Figure eA.17(b). At time t_d, the incident wave reaches point C at the end of the line, and the voltage rises there too. All of the current, I, flows into the resistor, Z_L, producing a voltage across the resistor of $Z_L I = Z_L (V_S/Z_0) = V_S$ because $Z_L = Z_0$. This voltage is consistent with the wave flowing along the transmission line. Thus, the wave is *absorbed* by the load impedance, and the transmission line reaches its *steady state*.

In steady state, the transmission line behaves like an ideal equipotential wire because it is, after all, just a wire. The voltage at all points along the line must be identical. Figure eA.18 shows the steady-state equivalent model of the circuit in Figure eA.16. The voltage is V_S everywhere along the wire.

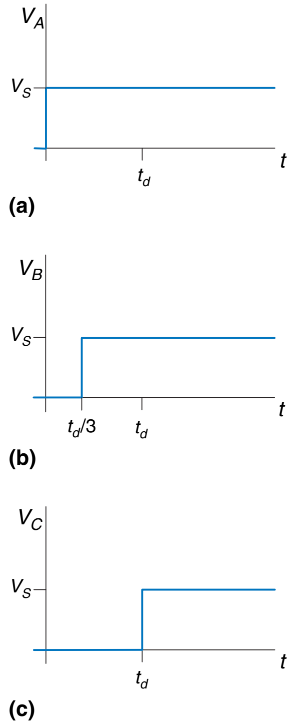


Figure eA.17 Voltage waveforms for Figure eA.16 at points *A*, *B*, and *C*

Example eA.2 TRANSMISSION LINE WITH MATCHED SOURCE AND LOAD TERMINATIONS

Figure eA.19 shows a transmission line with matched source and load impedances Z_S and Z_L. Plot the voltage at nodes A, B, and C versus time. When does the system reach steady-state, and what is the equivalent circuit at steady-state?

Solution: When the voltage source has a source impedance Z_S in series with the transmission line, part of the voltage drops across Z_S, and the remainder propagates down the transmission line. At first, the transmission line behaves as an impedance

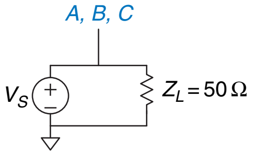


Figure eA.18 Equivalent circuit of Figure eA.16 at steady state

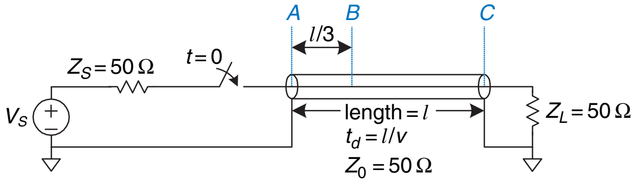


Figure eA.19 Transmission line with matched source and load impedances

Z_0, because the load at the end of the line cannot possibly influence the behavior of the line until a speed of light delay has elapsed. Hence, by the *voltage divider equation*, the incident voltage flowing down the line is

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{A.6}$$

Thus, at $t = 0$, a wave of voltage, $V = \frac{V_S}{2}$, is sent down the line from point A. Again, the signal reaches point B at time $t_d/3$ and point C at t_d, as shown in Figure eA.20. All of the current is absorbed by the load impedance Z_L, so the circuit enters steady-state at $t = t_d$. In steady-state, the entire line is at $V_S/2$, just as the steady-state equivalent circuit in Figure eA.21 would predict.

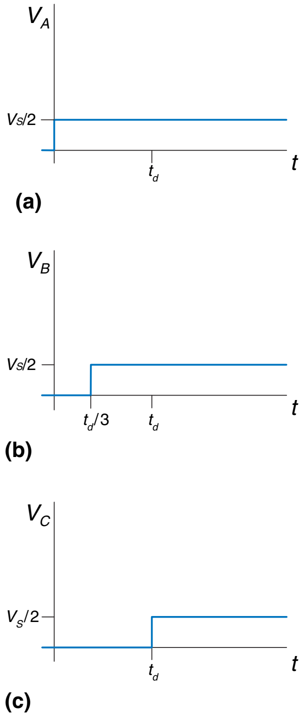


Figure eA.20 Voltage waveforms for Figure eA.19 at points *A*, *B*, and *C*

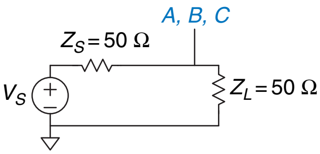


Figure eA.21 Equivalent circuit of Figure eA.19 at steady state

---

A.8.2 Open Termination

When the load impedance is not equal to Z_0, the termination cannot absorb all of the current, and some of the wave must be reflected. Figure eA.22 shows a transmission line with an open load termination. No current can flow through an open termination, so the current at point C must always be 0.

The voltage on the line is initially zero. At $t = 0$, the switch closes and a wave of voltage, $V = V_S\frac{Z_0}{Z_0 + Z_S} = \frac{V_S}{2}$, begins propagating down the line. Notice that this initial wave is the same as that of Example eA.2 and is independent of the termination, because the load at the end of the line cannot influence the behavior at the beginning until at least $2t_d$ has elapsed. This wave reaches point B at $t_d/3$ and point C at t_d as shown in Figure eA.23.

When the incident wave reaches point C, it cannot continue forward because the wire is open. It must instead reflect back toward the source. The reflected wave also has voltage $V = \frac{V_S}{2}$, because the open termination reflects the entire wave.

The voltage at any point is the sum of the incident and reflected waves. At time $t = t_d$, the voltage at point C is $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. The reflected wave reaches point B at $5t_d/3$ and point A at $2t_d$. When it reaches point A,

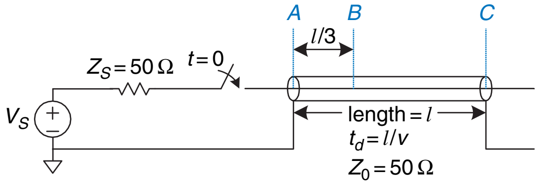


Figure eA.22 Transmission line with open load termination

the wave is absorbed by the source termination impedance that matches the characteristic impedance of the line. Thus, the system reaches steady state at time $t = 2t_d$, and the transmission line becomes equivalent to an equipotential wire with voltage V_S and current $I = 0$.

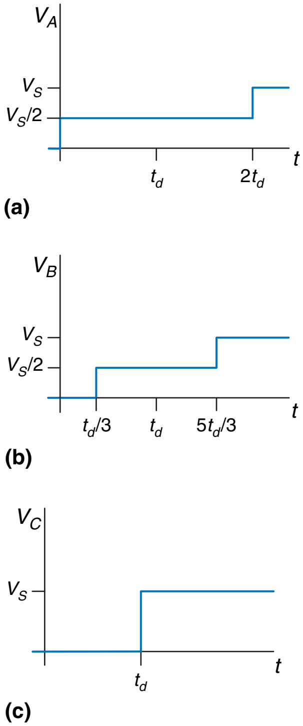


Figure eA.23 Voltage waveforms for Figure eA.22 at points *A*, *B*, and *C*

A.8.3 Short Termination

Figure eA.24 shows a transmission line terminated with a short circuit to ground. Thus, the voltage at point C must always be 0.

As in the previous examples, the voltages on the line are initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, begins propagating down the line (Figure eA.25). When it reaches the end of the line, it must reflect with opposite polarity. The reflected wave, with voltage $V = \frac{-V_S}{2}$, adds to the incident wave, ensuring that the voltage at point C remains 0. The reflected wave reaches the source at time $t = 2t_d$ and is absorbed by the source impedance. At this point, the system reaches steady state, and the transmission line is equivalent to an equipotential wire with voltage $V = 0$.

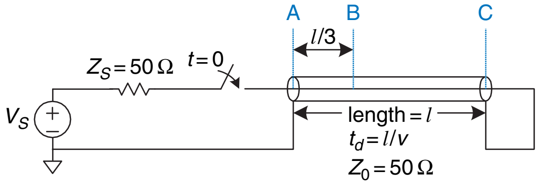


Figure eA.24 Transmission line with short termination

A.8.4 Mismatched Termination

The termination impedance is said to be *mismatched* when it does not equal the characteristic impedance of the line. In general, when an incident wave reaches a mismatched termination, part of the wave is absorbed and part is reflected. The reflection coefficient k_r indicates the fraction of the incident wave V_i that is reflected: $V_r = k_r V_i$.

Section A.8.8 derives the reflection coefficient using conservation of current arguments. It shows that, when an incident wave flowing along

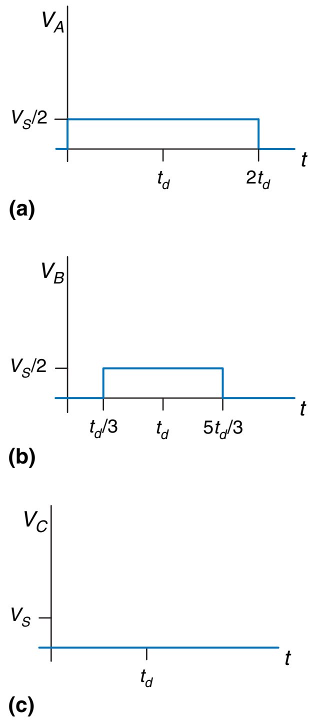


Figure eA.25 Voltage waveforms for Figure eA.24 at points *A*, *B*, and *C*

a transmission line of characteristic impedance Z_0 reaches a termination impedance Z_T at the end of the line, the reflection coefficient is

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0} \tag{A.7}$$

Note a few special cases. If the termination is an open circuit ($Z_T = \infty$), $k_r = 1$, because the incident wave is entirely reflected (so the current out the end of the line remains zero). If the termination is a short circuit ($Z_T = 0$), $k_r = -1$, because the incident wave is reflected with negative polarity (so the voltage at the end of the line remains zero). If the termination is a matched load ($Z_T = Z_0$), $k_r = 0$, because the incident wave is completely absorbed.

Figure eA.26 illustrates reflections in a transmission line with a *mismatched load termination* of 75 Ω. $Z_T = Z_L = 75\ \Omega$, and $Z_0 = 50\ \Omega$, so $k_r = 1/5$. As in previous examples, the voltage on the line is initially 0. When the switch closes, a wave of voltage $V = \frac{V_S}{2}$ propagates down the line, reaching the end at $t = t_d$. When the incident wave reaches the termination at the end of the line, one fifth of the wave is reflected, and the remaining four fifths flows into the load impedance. Thus, the reflected wave has a voltage $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. The total voltage at point C is the sum of the incoming and reflected voltages, $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. At $t = 2t_d$, the reflected wave reaches point *A*, where it is absorbed by the matched 50 Ω termination, Z_S. Figure eA.27 plots the voltages and currents along the line. Again, note that, in steady state (in this case at time $t > 2t_d$), the transmission line is equivalent to an equipotential wire, as shown in Figure eA.28. At steady state, the system acts like a voltage divider, so

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

Reflections can occur at both ends of the transmission line. Figure eA.29 shows a transmission line with a source impedance, Z_S, of 450 Ω and an open termination at the load. The reflection coefficients at the load and source, k_{rL} and k_{rS}, are 1 and 4/5, respectively. In this case, waves reflect off both ends of the transmission line until a steady state is reached.

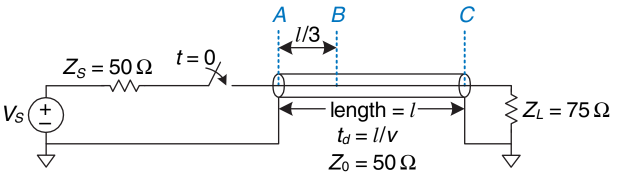


Figure eA.26 Transmission line with mismatched termination

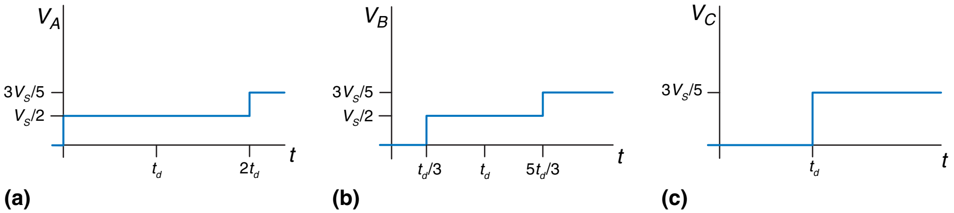


Figure eA.27 Voltage waveforms for Figure eA.26 at points *A*, *B*, and *C*

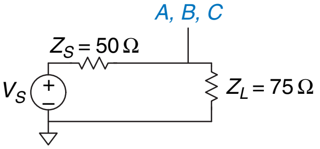

A, B, C
$Z_S = 50\ \Omega$
V_S
$Z_L = 75\ \Omega$

Figure eA.28 Equivalent circuit of Figure eA.26 at steady state

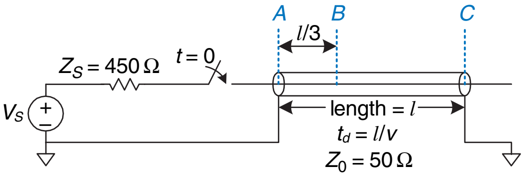

A
B
C
$l/3$
$t = 0$
$Z_S = 450\ \Omega$
V_S
length = l
$t_d = l/v$
$Z_0 = 50\ \Omega$

Figure eA.29 Transmission line with mismatched source and load terminations

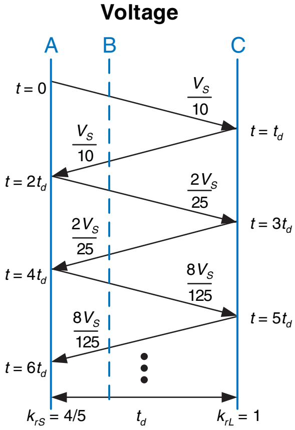

Voltage
A
B
C
$t = 0$
$\frac{V_S}{10}$
$t = t_d$
$\frac{V_S}{10}$
$t = 2t_d$
$\frac{2V_S}{25}$
$t = 3t_d$
$\frac{2V_S}{25}$
$t = 4t_d$
$\frac{8V_S}{125}$
$t = 5t_d$
$\frac{8V_S}{125}$
$t = 6t_d$
$k_{rS} = 4/5$
t_d
$k_{rL} = 1$

Figure eA.30 Bounce diagram for Figure eA.29

The *bounce diagram* shown in Figure eA.30 helps visualize reflections off both ends of the transmission line. The horizontal axis represents distance along the transmission line, and the vertical axis represents time, increasing downward. The two sides of the bounce diagram represent the source and load ends of the transmission line, points *A* and *C*. The incoming and reflected signal waves are drawn as diagonal lines between points *A* and *C*. At time $t = 0$, the source impedance and transmission line behave as a voltage divider, launching a voltage wave of $\frac{V_S}{10}$ from point *A* toward point *C*. At time $t = t_d$, the signal reaches point *C* and is completely reflected ($k_{rL} = 1$). At time $t = 2t_d$, the reflected wave of $\frac{V_S}{10}$ reaches point *A* and is reflected with a reflection coefficient, $k_{rS} = 4/5$, to produce a wave of $\frac{2V_S}{25}$ traveling toward point *C*, and so forth.

The voltage at a given time at any point on the transmission line is the sum of all the incident and reflected waves. Thus, at time $t = 1.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. At time $t = 3.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, and so forth. Figure eA.31 plots the voltages

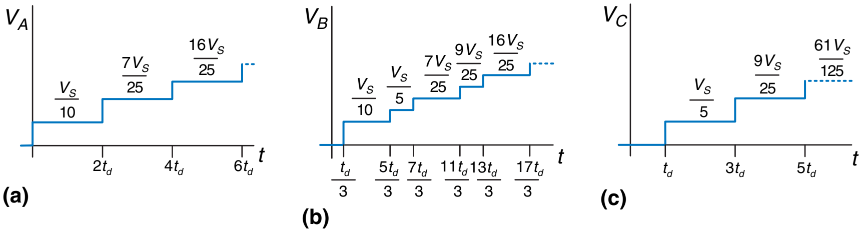


Figure eA.31 **Voltage and current waveforms for Figure eA.29**

against time. As t approaches infinity, the voltages approach steady state with $V_A = V_B = V_C = V_S$.

A.8.5 When to Use Transmission Line Models

Transmission line models for wires are needed whenever the wire delay, t_d, is longer than a fraction (e.g., 20%) of the edge rates (rise or fall times) of a signal. If the wire delay is shorter, it has an insignificant effect on the propagation delay of the signal, and the reflections dissipate while the signal is transitioning. If the wire delay is longer, it must be considered in order to accurately predict the propagation delay and waveform of the signal. In particular, reflections may distort the digital characteristic of a waveform, resulting in incorrect logic operations.

Recall that signals travel on a PCB at about 15 cm/ns. For TTL logic, with edge rates of 10 ns, wires must be modeled as transmission lines only if they are longer than 30 cm (10 ns × 15 cm/ns × 20%). PCB traces are usually less than 30 cm, so most traces can be modeled as ideal equipotential wires. In contrast, many modern chips have edge rates of 2 ns or less, so traces longer than about 6 cm (about 2.5 inches) must be modeled as transmission lines. Clearly, use of edge rates that are crisper than necessary just causes difficulties for the designer.

Breadboards lack a ground plane, so the electromagnetic fields of each signal are nonuniform and difficult to model. Moreover, the fields interact with other signals. This can cause strange reflections and crosstalk between signals. Thus, breadboards are unreliable above a few megahertz.

In contrast, PCBs have good transmission lines with consistent characteristic impedance and velocity along the entire line. As long as they are terminated with a source or load impedance that is matched to the impedance of the line, PCB traces do not suffer from reflections.

A.8.6 Proper Transmission Line Terminations

There are two common ways to properly terminate a transmission line, shown in Figure eA.32. In *parallel termination*, the driver has a low impedance ($Z_S \approx 0$). A load resistor Z_L with impedance Z_0 is placed in parallel

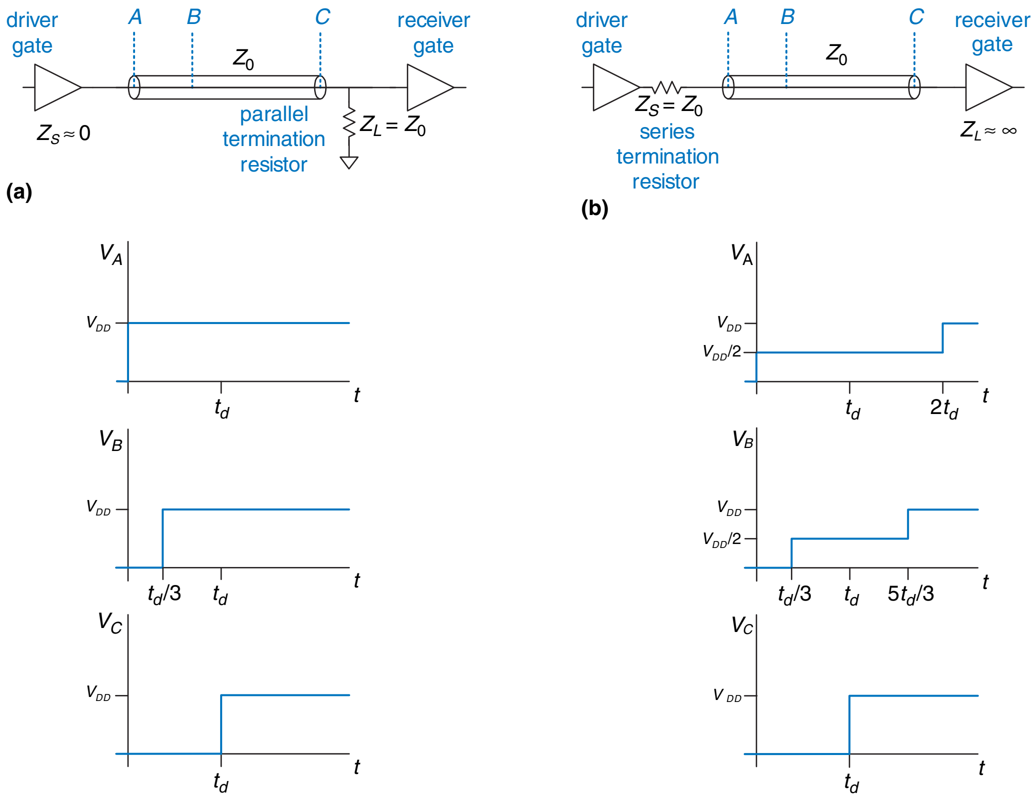

driver gate
A
B
C
Z_0
receiver gate
$Z_S \approx 0$
parallel termination resistor
$Z_L = Z_0$
(a)
driver gate
A
B
C
Z_0
receiver gate
$Z_S = Z_0$
series termination resistor
$Z_L \approx \infty$
(b)
V_A
V_{DD}
$V_{DD}/2$
V_B
V_C
$t_d/3$
t_d
$2t_d$
$5t_d/3$
t

Figure eA.32 Termination schemes: (a) parallel, (b) series

with the load (between the input of the receiver gate and ground). When the driver switches from 0 to V_{DD}, it sends a wave with voltage V_{DD} down the line. The wave is absorbed by the matched load termination, and no reflections take place. In *series termination*, a source resistor Z_S is placed in series with the driver to raise the source impedance to Z_0. The load has a high impedance ($Z_L \approx \infty$). When the driver switches, it sends a wave with voltage $V_{DD}/2$ down the line. The wave reflects at the open circuit load and returns, bringing the voltage on the line up to V_{DD}. The wave is absorbed at the source termination. Both schemes are similar in that the voltage at the receiver transitions from 0 to V_{DD} at $t = t_d$, just as one would desire. They differ in power consumption and in the waveforms that appear elsewhere along the line. Parallel termination dissipates power continuously through the load resistor when the line is at a high voltage. Series termination dissipates no DC power, because the load is an open circuit. However, in series terminated lines, points near the middle of the transmission line initially see a voltage of $V_{DD}/2$, until the reflection returns. If other gates are attached to the middle of the line, they will momentarily see an illegal logic level. Therefore, series termination works best for *point-to-point* communication with a single driver and a single receiver. Parallel termination is better for a *bus* with multiple receivers, because receivers at the middle of the line never see an illegal logic level.

A.8.7 Derivation of Z_0*

Z_0 is the ratio of voltage to current in a wave propagating along a transmission line. This section derives Z_0; it assumes some previous knowledge of resistor-inductor-capacitor (RLC) circuit analysis.

Imagine applying a step voltage to the input of a semi-infinite transmission line (so that there are no reflections). Figure eA.33 shows the semi-infinite line and a model of a segment of the line of length dx. R, L, and C, are the values of resistance, inductance, and capacitance per unit length. Figure eA.33(b) shows the transmission line model with a resistive component, R. This is called a *lossy* transmission line model, because energy is dissipated, or lost, in the resistance of the wire. However, this loss is often negligible, and we can simplify analysis by ignoring the resistive component and treating the transmission line as an *ideal* transmission line, as shown in Figure eA.33(c).

Voltage and current are functions of time and space throughout the transmission line, as given by Equations eA.8 and eA.9.

$$\frac{\partial}{\partial x}V(x, t) = L\frac{\partial}{\partial t}I(x, t) \tag{A.8}$$

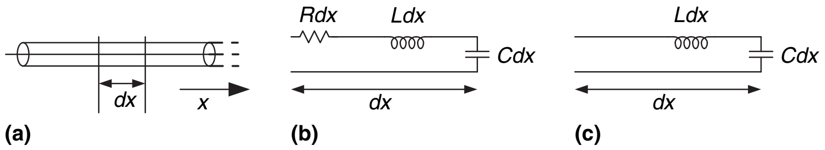
$$\frac{\partial}{\partial x}I(x, t) = C\frac{\partial}{\partial t}V(x, t) \tag{A.9}$$

Taking the space derivative of Equation eA.8 and the time derivative of Equation eA.9 and substituting gives Equation eA.10, the *wave equation*.

$$\frac{\partial^2}{\partial x^2}V(x, t) = LC\frac{\partial^2}{\partial t^2}V(x, t) \tag{A.10}$$

Z_0 is the ratio of voltage to current in the transmission line, as illustrated in Figure eA.34(a). Z_0 must be independent of the length of the line, because the behavior of the wave cannot depend on things at a distance. Because it is independent of length, the impedance must still equal Z_0 after the addition of a small amount of transmission line, dx, as shown in Figure eA.34(b).

Figure eA.33 **Transmission line models: (a) semi-infinite cable, (b) lossy, (c) ideal**



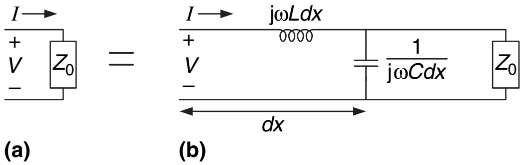


Figure eA.34 Transmission line model: (a) for entire line and (b) with additional length, *dx*

Using the impedances of an inductor and a capacitor, we rewrite the relationship of Figure eA.34 in equation form:

$$Z_0 = j\omega L dx + [Z_0 || (1/(j\omega C dx))] \tag{A.11}$$

Rearranging, we get

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 L C dx = 0 \tag{A.12}$$

Taking the limit as dx approaches 0, the last term vanishes and we find that

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.13}$$

A.8.8 Derivation of the Reflection Coefficient*

The reflection coefficient k_r is derived using conservation of current. Figure eA.35 shows a transmission line with characteristic impedance Z_0 and load impedance Z_L. Imagine an incident wave of voltage V_i and current I_i. When the wave reaches the termination, some current I_L flows through the load impedance, causing a voltage drop V_L. The remainder of the current reflects back down the line in a wave of voltage V_r and current I_r. Z_0 is the ratio of voltage to current in waves propagating along the line, so $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

The voltage on the line is the sum of the voltages of the incident and reflected waves. The current flowing in the positive direction on the line is the difference between the currents of the incident and reflected waves.

$$V_L = V_i + V_r \tag{A.14}$$

$$I_L = I_i - I_r \tag{A.15}$$

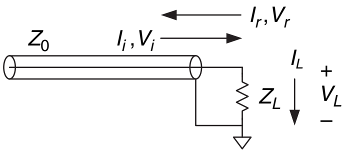


Figure eA.35 Transmission line showing incoming, reflected, and load voltages and currents

Using Ohm's law and substituting for I_L, I_i, and I_r in Equation eA.15, we get

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A.16}$$

Rearranging, we solve for the reflection coefficient, k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{A.17}$$

A.8.9 Putting It All Together

Transmission lines model the fact that signals take time to propagate down long wires because the speed of light is finite. An ideal transmission line has uniform inductance L and capacitance C per unit length and zero resistance. The transmission line is characterized by its characteristic impedance Z_0 and delay t_d which can be derived from the inductance, capacitance, and wire length. The transmission line has significant delay and noise effects on signals whose rise/fall times are less than about $5t_d$. This means that, for systems with 2 ns rise/fall times, PCB traces longer than about 6 cm must be analyzed as transmission lines to accurately understand their behavior.

A digital system consisting of a gate driving a long wire attached to the input of a second gate can be modeled with a transmission line as shown in Figure eA.36. The voltage source, source impedance Z_S, and switch model the first gate switching from 0 to 1 at time 0. The driver gate cannot supply infinite current; this is modeled by Z_S. Z_S is usually small for a logic gate, but a designer may choose to add a resistor in series with the gate to raise Z_S and match the impedance of the line. The input to the second gate is modeled as Z_L. CMOS circuits usually have little input current, so Z_L may be close to infinity. The designer may also choose to add a resistor in parallel with the second gate, between the gate input and ground, so that Z_L matches the impedance of the line.

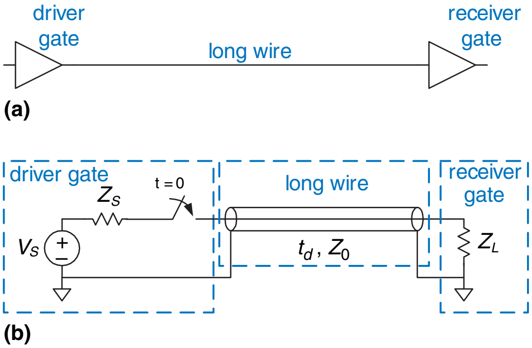


Figure eA.36 **Digital system modeled with transmission line**

When the first gate switches, a wave of voltage is driven onto the transmission line. The source impedance and transmission line form a voltage divider, so the voltage of the incident wave is

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A.18}$$

At time t_d, the wave reaches the end of the line. Part is absorbed by the load impedance, and part is reflected. The reflection coefficient k_r indicates the portion that is reflected: $k_r = V_r/V_i$, where V_r is the voltage of the reflected wave and V_i is the voltage of the incident wave.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A.19}$$

The reflected wave adds to the voltage already on the line. It reaches the source at time $2t_d$, where part is absorbed and part is again reflected. The reflections continue back and forth, and the voltage on the line eventually approaches the value that would be expected if the line were a simple equipotential wire.

A.9 ECONOMICS

Although digital design is so much fun that some of us would do it for free, most designers and companies intend to make money. Therefore, economic considerations are a major factor in design decisions.

The cost of a digital system can be divided into *nonrecurring engineering costs* (*NRE*), and *recurring costs*. NRE accounts for the cost of designing the system. It includes the salaries of the design team, computer and software costs, and the costs of producing the first working unit. The fully loaded cost of a designer in the United States in 2015 (including salary, health insurance, retirement plan, and a computer with design tools) was roughly $200,000 per year, so design costs can be significant. Recurring costs are the cost of each additional unit; this includes components, manufacturing, marketing, technical support, and shipping.

The sales price must cover not only the cost of the system but also other costs such as office rental, taxes, and salaries of staff who do not directly contribute to the design (such as the janitor and the CEO). After all of these expenses, the company should still make a profit.

Example eA.3 BEN TRIES TO MAKE SOME MONEY

Ben Bitdiddle has designed a crafty circuit for counting raindrops. He decides to sell the device and try to make some money, but he needs help deciding what implementation to use. He decides to use either an FPGA or an ASIC. The development kit to design and test the FPGA costs \$1500. Each FPGA costs \$17. The ASIC costs \$600,000 for a mask set and \$4 per chip.

Regardless of what chip implementation he chooses, Ben needs to mount the packaged chip on a printed circuit board (PCB), which will cost him \$1.50 per board. He thinks he can sell 1000 devices per month. Ben has coerced a team of bright undergraduates into designing the chip for their senior project, so it doesn't cost him anything to design.

If the sales price has to be twice the cost (100% profit margin), and the product life is 2 years, which implementation is the better choice?

Solution: Ben figures out the total cost for each implementation over 2 years, as shown in Table eA.4. Over 2 years, Ben plans on selling 24,000 devices, and the total cost is given in Table eA.4 for each option. If the product life is only two years, the FPGA option is clearly superior. The per-unit cost is \$445,500/24,000 = \$18.56, and the sales price is \$37.13 per unit to give a 100% profit margin. The ASIC option would have cost \$732,000/24,000 = \$30.50 and would have sold for \$61 per unit.

Table eA.4 **ASIC vs FPGA costs**

Cost	ASIC	FPGA
NRE	\$600,000	\$1500
chip	\$4	\$17
PCB	\$1.50	\$1.50
TOTAL	\$600,000 + (24,000 × \$5.50) = **\$732,000**	\$1500 + (24,000 × \$18.50) = **\$445,500**
per unit	\$30.50	\$18.56

Example eA.4 BEN GETS GREEDY

After seeing the marketing ads for his product, Ben thinks he can sell even more chips per month than originally expected. If he were to choose the ASIC option, how many devices per month would he have to sell to make the ASIC option more profitable than the FPGA option?

Solution: Ben solves for the minimum number of units, *N*, that he would need to sell in 2 years:

$$\$600,000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

Solving the equation gives $N = 46{,}039$ units, or 1919 units per month. He would need to almost double his monthly sales to benefit from the ASIC solution.

Example eA.5 BEN GETS LESS GREEDY

Ben realizes that his eyes have gotten too big for his stomach, and he doesn't think he can sell more than 1000 devices per month. But he does think the product life can be longer than 2 years. At a sales volume of 1000 devices per month, how long would the product life have to be to make the ASIC option worthwhile?

Solution: If Ben sells more than 46,039 units in total, the ASIC option is the best choice. So, Ben would need to sell at a volume of 1000 per month for at least 47 months (rounding up), which is almost 4 years. By then, his product is likely to be obsolete.

Chips are usually purchased from a distributor rather than directly from the manufacturer (unless you are ordering tens of thousands of units). Digikey (`www.digikey.com`) is a leading distributor that sells a wide variety of electronics. Jameco (`www.jameco.com`) and All Electronics (`www.allelectronics.com`) have eclectic catalogs that are competitively priced and well suited to hobbyists.

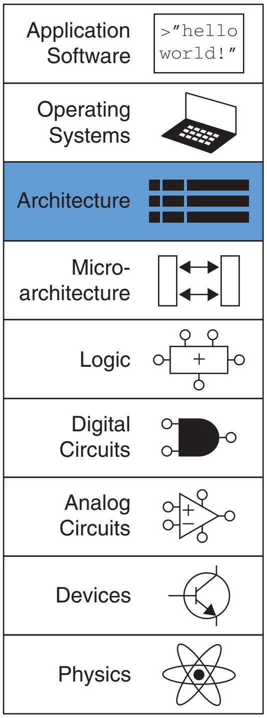
This page intentionally left blank

ARM Instructions

B

B.1 Data-processing Instructions
B.2 Memory Instructions
B.3 Branch Instructions
B.4 Miscellaneous Instructions
B.5 Condition Flags

This appendix summarizes ARMv4 instructions used in this book. Condition encodings are given in Table 6.3.


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
+
Digital Circuits
Analog Circuits
+
−
Devices
Physics

B.1 DATA-PROCESSING INSTRUCTIONS

Standard data-processing instructions use the encoding in Figure B.1. The 4-bit *cmd* field specifies the type of instruction as given in Table B.1. When the *S*-bit is 1, the status register is updated with the condition flags produced by the instruction. The *I*-bit and bits 4 and 7 specify one of three encodings for the second source operand, *Src2*, as described in Section 6.4.2. The *cond* field specifies which condition codes to check, as given in Section 6.3.2.

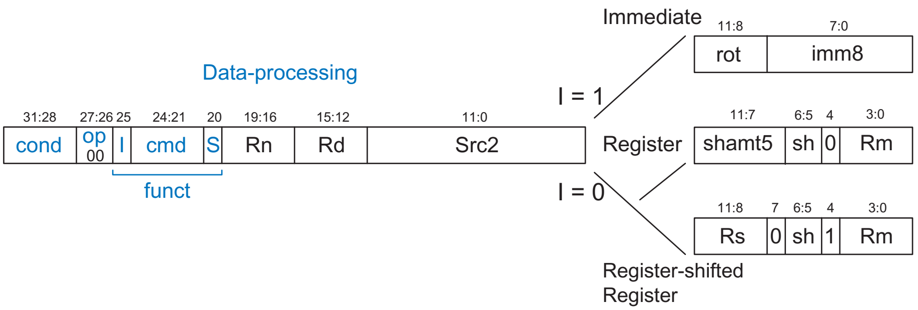

Data-processing
31:28
27:26
25
24:21
20
19:16
15:12
11:0
cond
op
00
I
cmd
S
Rn
Rd
Src2
funct
I = 1
I = 0
Immediate
11:8
7:0
rot
imm8
Register
11:7
6:5
4
3:0
shamt5
sh
0
Rm
11:8
7
6:5
4
3:0
Rs
0
sh
1
Rm
Register-shifted Register

Figure B.1 Data-processing instruction encodings

Table B.1 Data-processing instructions

cmd	Name	Description	Operation
0000	`AND Rd, Rn, Src2`	Bitwise AND	`Rd ← Rn & Src2`
0001	`EOR Rd, Rn, Src2`	Bitwise XOR	`Rd ← Rn ^ Src2`
0010	`SUB Rd, Rn, Src2`	Subtract	`Rd ← Rn - Src2`
0011	`RSB Rd, Rn, Src2`	Reverse Subtract	`Rd ← Src2 - Rn`
0100	`ADD Rd, Rn, Src2`	Add	`Rd ← Rn+Src2`
0101	`ADC Rd, Rn, Src2`	Add with Carry	`Rd ← Rn+Src2+C`
0110	`SBC Rd, Rn, Src2`	Subtract with Carry	`Rd ← Rn - Src2 -` $\overline{C}$
0111	`RSC Rd, Rn, Src2`	Reverse Sub w/ Carry	`Rd ← Src2 - Rn -` $\overline{C}$
1000 ($S = 1$)	`TST Rd, Rn, Src2`	Test	`Set flags based on Rn & Src2`
1001 ($S = 1$)	`TEQ Rd, Rn, Src2`	Test Equivalence	`Set flags based on Rn ^ Src2`
1010 ($S = 1$)	`CMP Rn, Src2`	Compare	`Set flags based on Rn - Src2`
1011 ($S = 1$)	`CMN Rn, Src2`	Compare Negative	`Set flags based on Rn+Src2`
1100	`ORR Rd, Rn, Src2`	Bitwise OR	`Rd ← Rn | Src2`
1101	Shifts:		
$I = 1$ OR ($\text{instr}_{11:4} = 0$)	`MOV Rd, Src2`	Move	`Rd ← Src2`
$I = 0$ AND ($sh = 00$; $\text{instr}_{11:4} \neq 0$)	`LSL Rd, Rm, Rs/shamt5`	Logical Shift Left	`Rd ← Rm << Src2`
$I = 0$ AND ($sh = 01$)	`LSR Rd, Rm, Rs/shamt5`	Logical Shift Right	`Rd ← Rm >> Src2`

(continued)

Table B.1 Data-processing instructions—Cont'd

cmd	Name	Description	Operation
$I = 0$ AND ($sh = 10$)	`ASR Rd, Rm, Rs/shamt5`	Arithmetic Shift Right	`Rd ← Rm>>>Src2`
$I = 0$ AND ($sh = 11$; $\text{instr}_{11:7,\ 4} = 0$)	`RRX Rd, Rm, Rs/shamt5`	Rotate Right Extend	`{Rd, C} ← {C, Rd}`
$I = 0$ AND ($sh = 11$; $\text{instr}_{11:7} \neq 0$)	`ROR Rd, Rm, Rs/shamt5`	Rotate Right	`Rd ← Rn ror Src2`
1110	`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd ← Rn & ~Src2`
1111	`MVN Rd, Rn, Src2`	Bitwise NOT	`Rd ← ~Rn`

`NOP` (no operation) is typically encoded as 0xE1A000, which is equivalent to `MOV R0, R0`.

B.1.1 Multiply Instructions

Multiply instructions use the encoding in Figure B.2 The 3-bit *cmd* field specifies the type of multiply, as given in Table B.2.

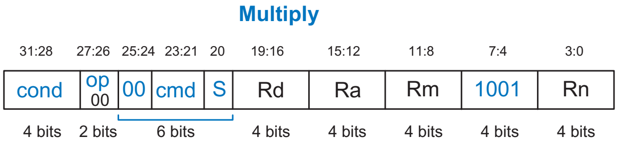


Figure B.2 Multiply instruction encoding

Table B.2 Multiply instructions

cmd	Name	Description	Operation
000	`MUL Rd, Rn, Rm`	Multiply	`Rd ← Rn × Rm (low 32 bits)`
001	`MLA Rd, Rn, Rm, Ra`	Multiply Accumulate	`Rd ← (Rn × Rm)+Ra (low 32 bits)`
100	`UMULL Rd, Rn, Rm, Ra`	Unsigned Multiply Long	`{Rd, Ra} ← Rn × Rm` `(all 64 bits, Rm/Rn unsigned)`
101	`UMLAL Rd, Rn, Rm, Ra`	Unsigned Multiply Accumulate Long	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra}` `(all 64 bits, Rm/Rn unsigned)`
110	`SMULL Rd, Rn, Rm, Ra`	Signed Multiply Long	`{Rd, Ra} ← Rn × Rm` `(all 64 bits, Rm/Rn signed)`
111	`SMLAL Rd, Rn, Rm, Ra`	Signed Multiply Accumulate Long	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra}` `(all 64 bits, Rm/Rn signed)`

B.2 MEMORY INSTRUCTIONS

The most common memory instructions (`LDR`, `STR`, `LDRB`, and `STRB`) operate on words or bytes and are encoded with *op* = 01. Extra memory instructions operating on halfwords or signed bytes are encoded with *op* = 00 and have less flexibility generating *Src2*. The immediate offset is only 8 bits and the register offset cannot be shifted. `LDRB` and `LDRH` zero-extend the bits to fill a word, while `LDRSB` and `LDRSH` sign-extend the bits. Also see memory indexing modes in Section 6.3.6.

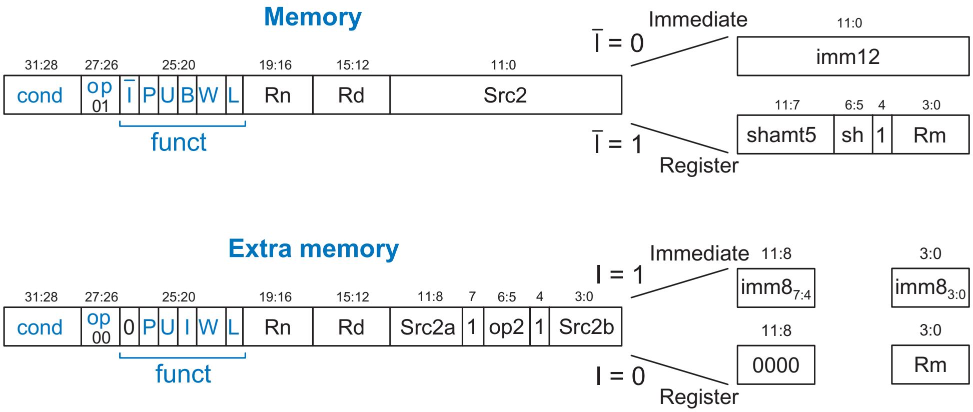

Memory
31:28
27:26
25:20
19:16
15:12
11:0
cond
op 01
Ī
P
U
B
W
L
Rn
Rd
Src2
funct
Ī = 0
Immediate
11:0
imm12
Ī = 1
Register
11:7
6:5
4
3:0
shamt5
sh
1
Rm
Extra memory
31:28
27:26
25:20
19:16
15:12
11:8
7
6:5
4
3:0
cond
op 00
0
P
U
I
W
L
Rn
Rd
Src2a
1
op2
1
Src2b
funct
I = 1
Immediate
11:8
imm8$_{7:4}$
3:0
imm8$_{3:0}$
I = 0
Register
11:8
0000
3:0
Rm

Figure B.3 Memory instruction encodings

Table B.3 Memory instructions

op	B	op2	L	Name	Description	Operation
01	0	N/A	0	`STR Rd, [Rn, ±Src2]`	Store Register	`Mem[Adr] ← Rd`
01	0	N/A	1	`LDR Rd, [Rn, ±Src2]`	Load Register	`Rd ← Mem[Adr]`
01	1	N/A	0	`STRB Rd, [Rn, ±Src2]`	Store Byte	`Mem[Adr] ← Rd`$_{7:0}$
01	1	N/A	1	`LDRB Rd, [Rn, ±Src2]`	Load Byte	`Rd ← Mem[Adr]`$_{7:0}$
00	N/A	01	0	`STRH Rd, [Rn, ±Src2]`	Store Halfword	`Mem[Adr] ← Rd`$_{15:0}$
00	N/A	01	1	`LDRH Rd, [Rn, ±Src2]`	Load Halfword	`Rd ← Mem[Adr]`$_{15:0}$
00	N/A	10	1	`LDRSB Rd, [Rn, ±Src2]`	Load Signed Byte	`Rd ← Mem[Adr]`$_{7:0}$
00	N/A	11	1	`LDRSH Rd, [Rn, ±Src2]`	Load Signed Half	`Rd ← Mem[Adr]`$_{15:0}$

B.3 BRANCH INSTRUCTIONS

Figure B.4 shows the encoding for branch instructions (`B` and `BL`) and Table B.4 describes their operation.

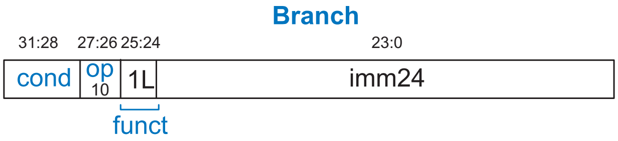


Figure B.4 **Branch instruction encoding**

Table B.4 **Branch instructions**

L	Name	Description	Operation
0	`B label`	Branch	`PC ← (PC+8)+imm24 << 2`
1	`BL label`	Branch with Link	`LR ← (PC+8) – 4; PC ← (PC+8)+imm24 << 2`

B.4 MISCELLANEOUS INSTRUCTIONS

The ARMv4 instruction set includes the following miscellaneous instructions. Consult the ARM Architecture Reference Manual for details.

Instructions	Description	Purpose
`LDM, STM`	Load/store multiple	Save and recall registers in subroutine calls
`SWP / SWPB`	Swap (byte)	Atomic load and store for process synchronization
`LDRT, LDRBT, STRT, STRBT`	Load/store word/byte with translation	Allow operating system to access memory in user virtual memory space
`SWI`[1]	Software Interrupt	Create an exception, often used to call the operating system
`CDP, LDC, MCR, MRC, STC`	Coprocessor access	Communicate with optional coprocessor
`MRS, MSR`	Move from/to status register	Save status register during exceptions

[1] `SWI` was renamed `SVC` (supervisor call) in ARMv7.

B.5 CONDITION FLAGS

Condition flags are changed by data-processing instructions with $S = 1$ in the machine code. All instructions except `CMP`, `CMN`, `TEQ`, and `TST` must have an "`S`" appended to the instruction mnemonic to make $S = 1$. Table B.5 shows which condition flags are affected by each instruction.

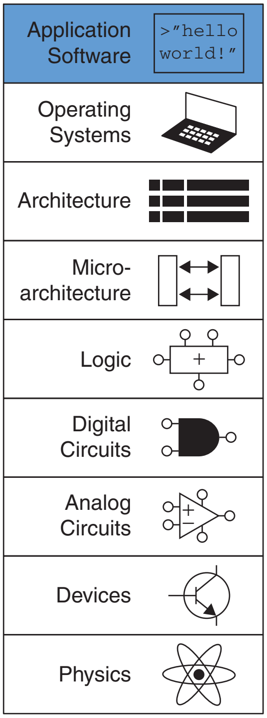
Table B.5 **Instructions that affect condition flags**

Type	Instructions	Condition Flags
Add	`ADDS, ADCS`	*N, Z, C, V*
Subtract	`SUBS, SBCS, RSBS, RSCS`	*N, Z, C, V*
Compare	`CMP, CMN`	*N, Z, C, V*
Shifts	`ASRS, LSLS, LSRS, RORS, RRXS`	*N, Z, C*
Logical	`ANDS, ORRS, EORS, BICS`	*N, Z, C*
Test	`TEQ, TST`	*N, Z, C*
Move	`MOVS, MVNS`	*N, Z, C*
Multiply	`MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS`	*N, Z*

C Programming

C

C.1 Introduction
C.2 Welcome to C
C.3 Compilation
C.4 Variables
C.5 Operators
C.6 Function Calls
C.7 Control-Flow Statements
C.8 More Data Types
C.9 Standard Libraries
C.10 Compiler and Command Line Options
C.11 Common Mistakes


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
+
Digital Circuits
Analog Circuits
+
−
Devices
Physics

C.1 INTRODUCTION

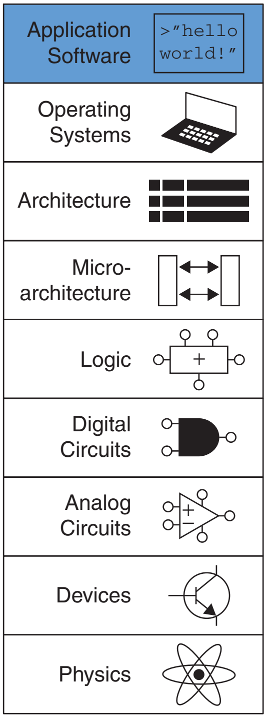
The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This Appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the Appendix so that readers may easily cover or skip it depending on previous experience.

The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.

eC

C Programming

C.1 Introduction
C.2 Welcome to C
C.3 Compilation
C.4 Variables
C.5 Operators
C.6 Function Calls
C.7 Control-Flow Statements
C.8 More Data Types
C.9 Standard Libraries
C.10 Compiler and Command Line Options
C.11 Common Mistakes



C.1 INTRODUCTION

The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This Appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the Appendix so that readers may easily cover or skip it depending on previous experience.

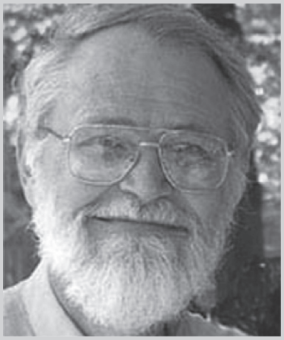
Programmers use many different languages to tell a computer what to do. Fundamentally, computers process instructions in *machine language* consisting of 1's and 0's, as is explored in Chapter 6. But programming in machine language is tedious and slow, leading programmers to use more abstract languages to get their meaning across more efficiently. Table eC.1 lists some examples of languages at various levels of abstraction.

One of the most popular programming languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C (including a family of closely related languages such as C++, C#, and Objective C) is the most widely used language in existence. Its popularity stems from a number of factors including its:

- Availability on a tremendous variety of platforms, from supercomputers down to embedded microcontrollers
- Relative ease of use, with a huge user base

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00017-3
© 2013 Elsevier, Inc. All rights reserved.

Dennis Ritchie, 1941–2011



Brian Kernighan, 1942–

C was formally introduced in 1978 by Brian Kernighan and Dennis Ritchie's classic book, *The C Programming Language*. In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89. Shortly thereafter, in 1990, this standard was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO/IEC updated the standard in 1999 to what is called C99, which we will be discussing in this text.

Table eC.1 **Languages at roughly decreasing levels of abstraction**

Language	Description
Matlab	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

- Moderate level of abstraction providing higher productivity than assembly language, yet giving the programmer a good understanding of how the code will be executed
- Suitability for generating high performance programs
- Ability to interact directly with the hardware

This chapter is devoted to C programming for a variety of reasons. Most importantly, C allows the programmer to directly access addresses in memory, illustrating the connection between hardware and software emphasized in this book. C is a practical language that all engineers and computer scientists should know. Its uses in many aspects of implementation and design – e.g., software development, embedded systems programming, and simulation – make proficiency in C a vital and marketable skill.

The following sections describe the overall syntax of a C program, discussing each part of the program including the header, function and variable declarations, data types, and commonly used functions provided in libraries. Chapter 9 (available as a web supplement, see Preface) describes a hands-on application by using C to program an ARM-based Raspberry Pi computer.

SUMMARY

- **High-level programming:** High-level programming is useful at many levels of design, from writing analysis or simulation software to programming microcontrollers that interact with hardware.
- **Low-level access:** C code is powerful because, in addition to high-level constructs, it provides access to low-level hardware and memory.

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer. C Code Example eC.1 is a simple C program that prints the phrase "Hello world!" to the *console*, the computer screen. C programs are generally contained in one or more text files that end in ".c". Good programming style requires a file name that indicates the contents of the program – for example, this file could be called `hello.c`.

C is the language used to program such ubiquitous systems as Linux, Windows, and iOS. C is a powerful language because of its direct access to hardware. As compared with other high level languages, for example Perl and Matlab, C does not have as much built-in support for specialized operations such as file manipulation, pattern matching, matrix manipulation, and graphical user interfaces. It also lacks features to protect the programmer from common mistakes, such as writing data past the end of an array. Its power combined with its lack of protection has assisted hackers who exploit poorly written software to break into computer systems.

C Code Example eC.1 SIMPLE C PROGRAM

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C.2.1 C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the `main` function, which is where the program starts executing. Most programs use other functions defined elsewhere in the C code and/or in a library. The overall sections of the `hello.c` program are the header, the `main` function, and the body.

Header: `#include <stdio.h>`
The header includes the *library functions* needed by the program. In this case, the program uses the `printf` function, which is part of the standard I/O library, `stdio.h`. See Section C.9 for further details on C's built-in libraries.

Main function: `int main(void)`
All C programs must include exactly one `main` function. Execution of the program occurs by running the code inside `main`, called the *body* of `main`. Function syntax is described in Section C.6. The body of a function contains a sequence of *statements*. Each statement ends with a semicolon. `int` denotes that the `main` function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

While this chapter provides a fundamental understanding of C programming, entire texts are written that describe C in depth. One of our favorites is the classic text *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of C. This text gives a concise description of the nuts and bolts of C. Another good text is *A Book on C* by Al Kelley and Ira Pohl.

Body: `printf("Hello world!\n");`
The body of this `main` function contains one statement, a call to the `printf` function, which prints the phrase “Hello world!” followed by a newline character indicated by the special sequence `"\n"`. Further details about I/O functions are described in Section C.9.1.

All programs follow the general format of the simple `hello.c` program. Of course, very complex programs may contain millions of lines of code and span hundreds of files.

C.2.2 Running a C Program

C programs can be run on many different machines. This *portability* is another advantage of C. The program is first compiled on the desired machine using the C *compiler*. Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler). Here we show how to compile and run a C program using gcc, which is freely available for download. It runs directly on Linux machines and is accessible under the Cygwin environment on Windows machines. It is also available for many embedded systems such as the ARM-based Raspberry Pi. The general process described below of C file creation, compilation, and execution is the same for any C program.

1. Create the text file, for example `hello.c`.
2. In a terminal window, change to the directory that contains the file `hello.c` and type `gcc hello.c` at the command prompt.
3. The compiler creates an executable file. By default, the executable is called `a.out` (or `a.exe` on Windows machines).
4. At the command prompt, type `./a.out` (or `./a.exe` on Windows) and press return.
5. “Hello world!” will appear on the screen.

SUMMARY

- `filename.c`: C program files are typically named with a .c extension.
- `main`: Each C program must have exactly one `main` function.
- `#include`: Most C programs use functions provided by built-in libraries. These functions are used by writing `#include <library.h>` at the top of the C file.
- `gcc filename.c`: C files are converted into an executable using a compiler such as the GNU compiler (gcc) or the C compiler (cc).
- **Execution:** After compilation, C programs are executed by typing `./a.out` (or `./a.exe`) at the command line prompt.

C.3 COMPILATION

A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable. Entire textbooks are written on compilers, but we describe them here briefly. The overall operation of the compiler is to (1) preprocess the file by including referenced libraries and expanding macro definitions, (2) ignore all unnecessary information such as comments, (3) translate the high-level code into simple instructions native to the processor that are represented in binary, called machine language, and (4) compile all the instructions into a single binary executable that can be read and executed by the computer. Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run. For example, the ARM machine language is covered in Chapter 6 in detail.

C.3.1 Comments

Programmers use comments to describe code at a high-level and clarify code function. Anyone who has read uncommented code can attest to their importance. C programs use two types of comments: Single-line comments begin with `//` and terminate at the end of the line; multiple-line comments begin with `/*` and end with `*/`. While comments are critical to the organization and clarity of a program, they are ignored by the compiler.

```
// This is an example of a one-line comment.

/* This is an example
   of a multi-line comment. */
```

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose. The comment below could be included at the top of the `hello.c` file.

```
// hello.c
// 1 Jan 2015 Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// This program prints "Hello world!" to the screen
```

C.3.2 `#define`

Constants are named using the `#define` directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you write a program that allows at most 5 user guesses, you can use `#define` to identify that number.

```
#define MAXGUESSES 5
```

Number constants in C default to decimal but can also be hexadecimal (prefix "0x") or octal (prefix "0"). Binary constants are not defined in C99 but are supported by some compilers (prefix "0b"). For example, the following assignments are equivalent:

```
char x = 37;
char x = 0x25;
char x = 045;
```

The # indicates that this line in the program will be handled by the *preprocessor*. Before compilation, the preprocessor replaces each occurrence of the identifier `MAXGUESSES` in the program with `5`. By convention, `#define` lines are located at the top of the file and identifiers are written in all capital letters. By defining constants in one location and then using the identifier in the program, the program remains consistent, and the value is easily modified – it need only be changed at the `#define` line instead of at each line in the code where the value is needed.

C Code Example eC.2 shows how to use the `#define` directive to convert inches to centimeters. The variables `inch` and `cm` are declared to be `float` to indicate they represent single-precision floating point numbers. If the conversion factor (`INCH2CM`) were used throughout a large program, having it declared using `#define` obviates errors due to typos (for example, typing 2.53 instead of 2.54) and makes it easy to find and change (for example, if more significant digits were required).

Globally defined constants eradicate *magic numbers* from a program. A magic number is a constant that shows up in a program without a name. The presence of magic numbers in a program often introduces tricky bugs – for example, when the number is changed in one location but not another.

C Code Example eC.2 USING `#define` TO DECLARE CONSTANTS

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

C.3.3 #include

Modularity encourages us to split programs across separate files and functions. Commonly used functions can be grouped together for easy reuse. Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the `#include` preprocesser directive. *Standard libraries* that provide commonly used functions are accessed in this way. For example, the following line is required to use the functions defined in the standard input/output (I/O) library, such as `printf`.

```
#include <stdio.h>
```

The "`.h`" postfix of the include file indicates it is a header file. While `#include` directives can be placed anywhere in the file before the included functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included by using quotation marks (`" "`) around the file name instead of brackets (`< >`). For example, a user-created header file called `myfunctions.h` would be included using the following line.

```
#include "myfunctions.h"
```

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included.

SUMMARY

- **Comments:** C provides single-line comments (`//`) and multi-line comments (`/* */`).
- **`#define NAME val`:** the `#define` directive allows an identifier (`NAME`) to be used throughout the program. Before compilation, all instances of `NAME` are replaced with `val`.
- **`#include`:** `#include` allows common functions to be used in a program. For built-in libraries, include the following line at the top of the code: `#include <library.h>` To include a user-defined header file, the name must be in quotes, listing the path relative to the current directory as needed: i.e., `#include "other/myFuncs.h"`.

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example, the following declaration states that the variable is of type `char` (which is a 1-byte type), and that the variable name is `x`. The compiler decides where to place this 1-byte variable in memory.

```
char x;
```

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure eC.1. A variable occupies one or more bytes of memory, and the address of multiple-byte variables is indicated by the lowest numbered byte. The type of a variable indicates whether to interpret the byte(s) as an integer, floating point number, or other type. The rest of this section describes C's primitive data types, the declaration of global and local variables, and the initialization of variables.

Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., `int`, `while`, etc.), may not start with a number (i.e., `int 1x;` is not a valid declaration), and may not include special characters such as `\`, `*`, `?`, or `-`. Underscores (_) are allowed.

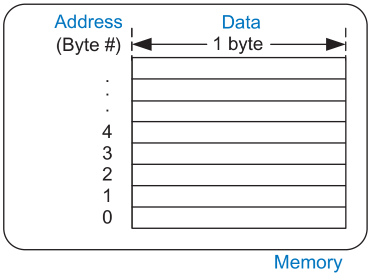


Figure eC.1 C's view of memory

C.4.1 Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized as integers, floating-point variables, and characters. An integer represents a two's complement or unsigned number within a finite range. A floating-point variable uses IEEE floating point representation to describe real numbers with a finite range and precision. A character can be viewed as either an ASCII value or an 8-bit integer.[1] Table eC.2 lists the size and range of each primitive type. Integers may be 16, 32, or 64 bits. They use two's complement unless qualified as `unsigned`.

Table eC.2 Primitive data types and sizes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

---

[1] Technically, the C99 standard defines a character as "a bit representation that fits in a byte," without requiring a byte to be 8 bits. However, current systems define a byte as 8 bits.

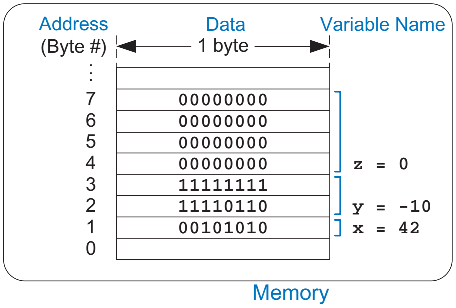


Figure eC.2 **Variable storage in memory for C Code Example eC.3**

The size of the `int` type is machine dependent and is generally the native word size of the machine. For example, on a 32-bit ARM processor, the size of an `int` or `unsigned int` is 32 bits. Floating point numbers may be 32- or 64-bit single or double precision. Characters are 8 bits.

C Code Example eC.3 shows the declaration of variables of different types. As shown in Figure eC.2, `x` requires one byte of data, `y` requires two, and `z` requires four. The program decides where these bytes are stored in memory, but each type always requires the same amount of data. For illustration, the addresses of `x`, `y`, and `z` in this example are 1, 2, and 4. Variable names are case-sensitive, so, for example, the variable `x` and the variable `X` are two different variables. (But it would be very confusing to use both in the same program!)

The machine-dependent nature of the `int` data type is a blessing and a curse. On the bright side, it matches the native word size of the processor so it can be fetched and manipulated efficiently. On the down side, programs using `int`s may behave differently on different computers. For example, a banking program might store the number of cents in your bank account as an `int`. When compiled on a 64-bit PC, it will have plenty of range for even the wealthiest entrepreneur. But if it is ported to a 16-bit microcontroller, it will overflow for accounts exceeding $327.67, resulting in unhappy and poverty-stricken customers.

C Code Example eC.3 EXAMPLE DATA TYPES

```
// Examples of several data types and their binary representations
unsigned char x = 42;        // x = 00101010
short y = −10;               // y = 11111111 11110110
unsigned long z = 0;         // z = 00000000 00000000 00000000 00000000
```

C.4.2 Global and Local Variables

Global and local variables differ in where they are declared and where they are visible. A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions. Global variables should be used sparingly because they violate the principle of modularity, making large programs more difficult to read. However, a variable accessed by many functions can be made global.

A local variable is declared inside a function and can only be used by that function. Therefore, two functions can have local variables with the same names without interfering with each other. Local variables are declared at the beginning of a function. They cease to exist when the function ends and are recreated when the function is called again. They do not retain their value from one invocation of a function to the next.

The *scope* of a variable is the context in which it can be used. For example, for a local variable, its scope is the function in which it is declared. It is out of scope everywhere else.

C Code Examples eC.4 and eC.5 compare programs using global versus local variables. In C Code Example eC.4, the global variable `max` can be accessed by any function. Using a local variable, as shown in C Code Example eC.5, is the preferred style because it preserves the well-defined interface of modularity.

C Code Example eC.4 GLOBAL VARIABLES

```
// Use a global variable to find and print the maximum of 3 numbers

int max;              // global variable holding the maximum value

void findMax(int a, int b, int c) {
  max = a;
  if (b > max) {
    if (c > b) max = c;
    else       max = b;
  } else if (c > max) max = c;
}

void printMax(void) {
  printf("The maximum number is: %d\n", max);
}

int main(void) {
  findMax(4, 3, 7);
  printMax();
}
```

C Code Example eC.5 LOCAL VARIABLES

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
  int result = a;  // local variable holding the maximum value

  if (b > result) {
    if (c > b) result = c;
    else       result = b;
  } else if (c > result) result = c;

  return result;
}

void printMax(int m) {
  printf("The maximum number is: %d\n", m);
}

int main(void) {
  int max;

  max = getMax(4, 3, 7);
  printMax(max);
}
```

C.4.3 Initializing Variables

A variable needs to be *initialized* – assigned a value – before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program. C Code Example eC.3 shows variables initialized at the same time they are declared. C Code Example eC.4 shows how variables are initialized before their use, but after declaration; the global variable `max` is initialized by the `getMax` function before it is read by the `printMax` function. Reading from uninitialized variables is a common programming error, and can be tricky to debug.

SUMMARY

- **Variables:** Each variable is defined by its data type, name, and memory location. A variable is declared as `datatype name`.
- **Data types:** A data type describes the size (number of bytes) and representation (interpretation of the bytes) of a variable. Table eC.2 lists C's built-in data types.
- **Memory:** C views memory as a list of bytes. Memory stores variables and associates each variable with an address (byte number).
- **Global variables:** Global variables are declared outside of all functions and can be accessed anywhere in the program.
- **Local variables:** Local variables are declared within a function and can be accessed only within that function.
- **Variable initialization:** Each variable must be initialized before it is read. Initialization can happen either at declaration or afterward.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

```
y = a + 3;
```

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants. C supports the operators shown in Table eC.3, listed by category and in order of decreasing precedence. For example, multiplicative operators take precedence over additive

Table eC.3 **Operators listed by decreasing precedence**

Category	Operator	Description	Example
Unary	++	post-increment	a++; // a = a+1
	--	post-decrement	x--; // x = x-1

	&	memory address of a variable	x = &y; // x = the memory
			// address of y

	~	bitwise NOT	z = ~a;
	!	Boolean NOT	!x
	-	negation	y = -a;
	++	pre-increment	++a; // a = a+1
	--	pre-decrement	--x; // x = x-1

	(type)	casts a variable to (type)	x = (int)c; // cast c to an
			// int and assign it to x

	sizeof()	size of a variable or type in bytes	long int y; x = sizeof(y); // x = 4
Multiplicative	*	multiplication	y = x * 12;
	/	division	z = 9 / 3; // z = 3
	%	modulo	z = 5 % 2; // z = 1
Additive	+	addition	y = a + 2;
	-	subtraction	y = a - 2;
Bitwise Shift	<<	bitshift left	z = 5 << 2; // z = 0b00010100
	>>	bitshift right	x = 9 >> 3; // x = 0b00000001
Relational	==	equals	y == 2
	!=	not equals	x != 7
	<	less than	y < 12
	>	greater than	val > max
	<=	less than or equal	z <= 2
	>=	greater than or equal	y >= 10

(*continued*)

Table eC.3 Operators listed by decreasing precedence—Cont'd

Category	Operator	Description	Example
Bitwise	`&`	bitwise AND	`y = a & 15;`
	`^`	bitwise XOR	`y = 2 ^ 3;`
	`|`	bitwise OR	`y = a | b;`
Logical	`&&`	Boolean AND	`x && y`
	`||`	Boolean OR	`x || y`

Ternary	`? :`	ternary operator	`y = x ? a : b; // if x is TRUE,`
			`// y=a, else y=b`

Assignment	`=`	assignment	`x = 22;`
	`+=`	addition and assignment	`y += 3;     // y = y + 3`
	`-=`	subtraction and assignment	`z -= 10;    // z = z - 10`
	`*=`	multiplication and assignment	`x *= 4;     // x = x * 4`
	`/=`	division and assignment	`y /= 10;    // y = y / 10`
	`%=`	modulo and assignment	`x %= 4;     // x = x % 4`
	`>>=`	bitwise right-shift and assignment	`x >>= 5;    // x = x>>5`
	`<<=`	bitwise left-shift and assignment	`x <<= 2;    // x = x<<2`
	`&=`	bitwise AND and assignment	`y &= 15;    // y = y & 15`
	`|=`	bitwise OR and assignment	`x |= y;     // x = x | y`
	`^=`	bitwise XOR and assignment	`x ^= y;     // x = x ^ y`

operators. Within the same category, operators are evaluated in the order that they appear in the program.

Unary operators, also called monadic operators, have a single operand. Ternary operators have three operands, and all others have two. The ternary operator (from the Latin *ternarius* meaning consisting of three) chooses the second or third operand depending on whether the first value is TRUE (nonzero) or FALSE (zero), respectively. C Code Example eC.6 shows how to compute `y = max(a,b)` using the ternary operator, along with an equivalent but more verbose `if/else` statement.

The Truth, the Whole Truth, and Nothing But the Truth
C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as `if` and `while`, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

C Code Example eC.6 (a) TERNARY OPERATOR, AND (b) EQUIVALENT `if/else` STATEMENT

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer
(b) if (a > b) y = a;
    else       y = b;
```

Simple assignment uses the = operator. C code also allows for compound assignment, that is, assignment after a simple operation such as addition (+=) or multiplication (*=). In compound assignments, the variable on the left side is both operated on and assigned the result. C Code Example eC.7 shows these and other C operations. Binary values in the comments are indicated with the prefix “`0b`”.

C Code Example eC.7 OPERATOR EXAMPLES

Expression	Result	Notes
`44 / 14`	`3`	Integer division truncates
`44 % 14`	`2`	44 mod 14
`0x2C && 0xE   //0b101100 && 0b1110`	`1`	Logical AND
`0x2C || 0xE   //0b101100 || 0b1110`	`1`	Logical OR
`0x2C & 0xE    //0b101100 & 0b1110`	`0xC  (0b001100)`	Bitwise AND
`0x2C | 0xE    //0b101100 | 0b1110`	`0x2E (0b101110)`	Bitwise OR
`0x2C ^ 0xE    //0b101100 ^ 0b1110`	`0x22 (0b100010)`	Bitwise XOR
`0xE << 2      //0b1110 << 2`	`0x38 (0b111000)`	Left shift by 2
`0x2C >> 3     //0b101100 >> 3`	`0x5  (0b101)`	Right shift by 3
`x = 14;` `x += 2;`	`x=16`	

`y = 0x2C;     // y = 0b101100`
`y &= 0xF;     // y &= 0b1111`
	`y=0xC (0b001100)`	

`x = 14; y = 44;` `y = y + x++;`	`x=15, y=58`	Increment `x` after using it
`x = 14; y = 44;` `y = y + ++x;`	`x=15, y=59`	Increment `x` before using it

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example eC.8 shows the `sum3` function. The function declaration begins with the return type, `int`, followed by the name, `sum3`, and the inputs enclosed within parentheses `(int a, int b, int c)`. Curly braces `{}` enclose the body of the function, which may contain zero or more statements. The `return` statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can only return a single value.

C Code Example eC.8 `sum3` **FUNCTION**

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

After the following call to `sum3`, `y` holds the value 42.

```
int y = sum3(10, 15, 17);
```

Although a function may have inputs and outputs, neither is required. C Code Example eC.9 shows a function with no inputs or outputs. The keyword `void` before the function name indicates that nothing is returned. `void` between the parentheses indicates that the function has no input arguments.

Nothing between the parentheses also indicates no input arguments. So, in this case we could have written:

```
void printPrompt()
```

C Code Example eC.9 **FUNCTION** `printPrompt` **WITH NO INPUTS OR OUTPUTS**

```
// Print a prompt to the console
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

A function must be declared in the code before it is called. This may be done by placing the called function earlier in the file. For this reason, `main` is often placed at the end of the C file after all the functions it calls. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line of the function, declaring the return type, function name, and function inputs. For example, the function prototypes for the functions in C Code Examples eC.8 and eC.9 are:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

C Code Example eC.10 shows how function prototypes are used. Even though the functions themselves are after `main`, the function prototypes at the top of the file allow them to be used in `main`.

With careful ordering of functions, prototypes may be unnecessary. However, they are unavoidable in certain cases, such as when function `f1` calls `f2` and `f2` calls `f1`. It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

C Code Example eC.10 FUNCTION PROTOTYPES

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
{
  int y = sum3(10, 15, 20);

  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c) {
  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names so you don't have to constantly look up the correct capitalization. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., `printPrompt`), or to use underscores between words (e.g., `print_prompt`). We have unscientifically observed that reaching for the underscore key exacerbates carpal tunnel syndrome (my pinky finger twinges just thinking about the underscore!) and hence prefer camelCase. But the most important thing is to be consistent in style within your organization.

The `main` function is always declared to return an `int`, which conveys to the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If `main` reaches the end without encountering a `return` statement, it will automatically return 0. Most operating systems do not automatically inform the user of the value returned by the program.

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

C.7.1 Conditional Statements

`if`, `if/else`, and `switch/case` statements are conditional statements commonly used in high-level languages including C.

if Statements

An `if` statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
  statement
```

C Code Example eC.11 shows how to use an `if` statement in C. When the variable `aintBroke` is equal to 1, the variable `dontFix` is set to 1. A block of multiple statements can be executed by placing curly braces `{}` around the statements, as shown in C Code Example eC.12.

Curly braces, `{}`, are used to group one or more statements into a *compound statement* or *block*.

C Code Example eC.11 `if` STATEMENT

```
int dontFix = 0;
if (aintBroke == 1)
  dontFix = 1;
```

C Code Example eC.12 `if` STATEMENT WITH A BLOCK OF CODE

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

if/else Statements

`if/else` statements execute one of two statements depending on a condition, as shown below. When the `expression` in the `if` statement is TRUE, `statement1` is executed. Otherwise, `statement2` is executed.

```
if (expression)
  statement1
else
  statement2
```

C Code Example eC.6(b) gives an example `if/else` statement in C. The code sets `y` equal to `a` if `a` is greater than `b`; otherwise `y = b`.

switch/case Statements

`switch/case` statements execute one of several statements depending on the conditions, as shown in the general format below.

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
  default:            statement4
}
```

For example, if `variable` is equal to `expression2`, execution continues at `statement2` until the keyword `break` is reached, at which point it exits the `switch/case` statement. If no conditions are met, the `default` executes.

If the keyword `break` is omitted, execution begins at the point where the condition is TRUE and then falls through to execute the remaining cases below it. This is usually not what you want and is a common error among beginning C programmers.

C Code Example eC.13 shows a `switch/case` statement that, depending on the variable `option`, determines the amount of money `amt` to be disbursed. A `switch/case` statement is equivalent to a series of nested `if/else` statements, as shown by the equivalent code in C Code Example eC.14.

C Code Example eC.13 `switch/case` STATEMENT

```
// Assign amt depending on the value of option
switch (option) {
  case 1:  amt = 100; break;
  case 2:  amt = 50;  break;
  case 3:  amt = 20;  break;
  case 4:  amt = 10;  break;
  default: printf("Error: unknown option.\n");
}
```

C Code Example eC.14 NESTED `if/else` STATEMENT

```
// Assign amt depending on the value of option
if      (option == 1)  amt = 100;
else if (option == 2)  amt = 50;
else if (option == 3)  amt = 20;
else if (option == 4)  amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2 Loops

`while`, `do/while`, and `for` loops are common loop constructs used in many high-level languages including C. These loops repeatedly execute a statement as long as a condition is satisfied.

while Loops

`while` loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

```
while (condition)
  statement
```

The `while` loop in C Code Example eC.15 computes the factorial of 9 = 9 × 8 × 7 × ⋯ × 1. Note that the condition is checked before executing the `statement`. In this example, the statement is a compound statement or block, so curly braces are required.

C Code Example eC.15 `while` LOOP

```
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;

// multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

---

do/while Loops

`do/while` loops are like `while` loops but the `condition` is checked only after the `statement` is executed once. The general format is shown below. The `condition` is followed by a semi-colon.

```
do
  statement
while (condition);
```

The `do/while` loop in C Code Example eC.16 queries a user to guess a number. The program checks the condition (if the user's number is equal to the correct number) only after the body of the `do/while` loop executes once. This construct is useful when, as in this case, something must be done (for example, the guess retrieved from the user) before the condition is checked.

C Code Example eC.16 `do/while` LOOP

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
```

```
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);      // read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
   printf("You guessed the correct number!\n");
```

`for` Loops

`for` loops, like `while` and `do/while` loops, repeatedly execute a `statement` until a condition is not satisfied. However, `for` loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the `for` loop is

```
for (initialization; condition; loop operation)
  statement
```

The `initialization` code executes only once, before the `for` loop begins. The `condition` is tested at the beginning of each iteration of the loop. If the `condition` is not TRUE, the loop exits. The `loop operation` executes at the end of each iteration. C Code Example eC.17 shows the factorial of 9 computed using a `for` loop.

C Code Example eC.17 `for` LOOP

```
// Compute 9!
int i;    // loop variable
int fact = 1;

for (i=1; i<10; i++)
   fact *= i;
```

Whereas the `while` and `do/while` loops in C Code Examples eC.15 and eC.16 include code for incrementing and checking the loop variable `i` and `numGuesses`, respectively, the `for` loop incorporates those statements into its format. A `for` loop could be expressed equivalently, but less conveniently, as

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

SUMMARY

- **Control-flow statements:** C provides control-flow statements for conditional statements and loops.
- **Conditional statements:** Conditional statements execute a statement when a condition is TRUE. C includes the following conditional statements: `if`, `if/else`, and `switch/case`.
- **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides `while`, `do/while`, and `for` loops.

C.8 MORE DATA TYPES

Beyond various sizes of integers and floating-point numbers, C includes other special data types including pointers, arrays, strings, and structures. These data types are introduced in this section along with dynamic memory allocation.

C.8.1 Pointers

A pointer is the address of a variable. C Code Example eC.18 shows how to use pointers. `salary1` and `salary2` are variables that can contain integers, and `ptr` is a variable that can hold the address of an integer. The compiler will assign arbitrary locations in RAM for these variables depending on the runtime environment. For the sake of concreteness, suppose this program is compiled on a 32-bit system with `salary1` at addresses 0x70-73, `salary2` at addresses 0x74-77, and `ptr` at 0x78-7B. Figure eC.3 shows memory and its contents after the program is executed.

In a variable declaration, a star (`*`) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the `*` operator *dereferences* a pointer, returning the value stored at the

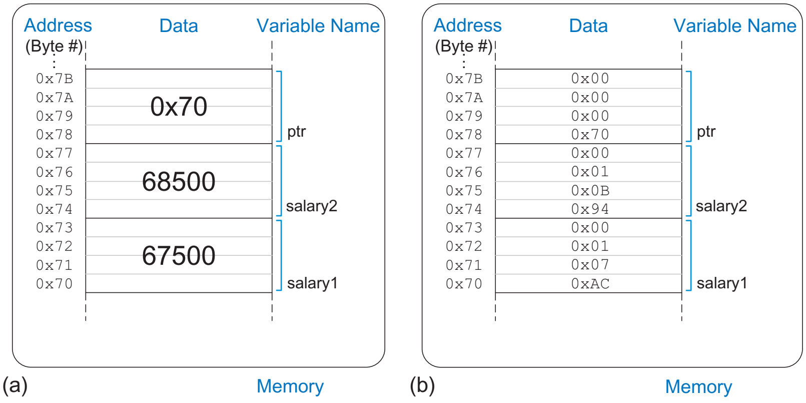

Address
(Byte #)
Data
Variable Name
0x7B
0x7A
0x79
0x78
0x77
0x76
0x75
0x74
0x73
0x72
0x71
0x70
0x70
68500
67500
ptr
salary2
salary1
(a)
Memory
Address
(Byte #)
Data
Variable Name
0x7B 0x00
0x7A 0x00
0x79 0x00
0x78 0x70
0x77 0x00
0x76 0x01
0x75 0x0B
0x74 0x94
0x73 0x00
0x72 0x01
0x71 0x07
0x70 0xAC
ptr
salary2
salary1
(b)
Memory

Figure eC.3 Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

Dereferencing a pointer to a non-existent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

indicated memory address contained in the pointer. The `&` operator is pronounced "address of," and it produces the memory address of the variable being referenced.

Pointers are particularly useful when a function needs to modify a variable, instead of just returning a value. Because functions can't modify their inputs directly, a function can make the input a pointer to the variable. This is called passing an input variable *by reference* instead of *by value*, as shown in prior examples. C Code Example eC.19 gives an example of passing `x` by reference so that `quadruple` can modify the variable directly.

C Code Example eC.18 POINTERS

```
// Example pointer manipulations
int salary1, salary2;  // 32-bit numbers
int *ptr;              // a pointer specifying the address of an int variable

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

C Code Example eC.19 PASSING AN INPUT VARIABLE BY REFERENCE

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a)
{
  *a = *a * 4;
}

int main(void)
{
  int x = 5;

  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as `NULL` in a program.

C.8.2 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to *N*−1, where *N* is the length of the array. C Code Example eC.20 declares an array variable called `scores` that holds the final exam scores for three students. Memory space is reserved for three `long`s, that is, 3 × 4 = 12 bytes. Suppose the `scores` array starts at address 0x40. The address of the 1st element (i.e., `scores[0]`) is 0x40, the 2nd element is 0x44, and the 3rd element is 0x48, as shown in Figure eC.4. In C, the array variable, in this case `scores`, is a pointer to the 1st element. It is the programmer's responsibility not to access elements beyond the end of the array. C has no internal bounds checking, so a program that writes beyond the end of an array will compile fine but may stomp on other parts of memory when it runs.

C Code Example eC.20 ARRAY DECLARATION

```
long scores[3];    // array of three 4-byte numbers
```

The elements of an array can be initialized either at declaration using curly braces `{}`, as shown in C Code Example eC.21, or individually in the body of the code, as shown in C Code Example eC.22. Each element of an array is accessed using brackets `[]`. The contents of memory containing the array are shown in Figure eC.4. Array initialization using curly braces `{}` can only be performed at declaration, and not afterward. `for` loops are commonly used to assign and read array data, as shown in C Code Example eC.23.

C Code Example eC.21 ARRAY INITIALIZATION AT DECLARATION USING { }

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

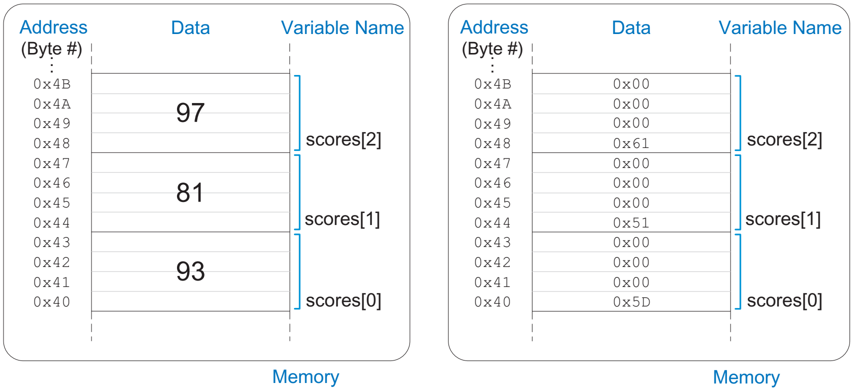

Address
(Byte #)
Data
Variable Name
0x4B
0x4A
0x49
0x48
0x47
0x46
0x45
0x44
0x43
0x42
0x41
0x40
97
81
93
scores[2]
scores[1]
scores[0]
Memory
Address
(Byte #)
Data
Variable Name
0x4B 0x00
0x4A 0x00
0x49 0x00
0x48 0x61
0x47 0x00
0x46 0x00
0x45 0x00
0x44 0x51
0x43 0x00
0x42 0x00
0x41 0x00
0x40 0x5D
scores[2]
scores[1]
scores[0]
Memory

Figure eC.4 `scores` **array stored in memory**

C Code Example eC.22 ARRAY INITIALIZATION USING ASSIGNMENT

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example eC.23 ARRAY INITIALIZATION USING A `for` LOOP

```
// User enters 3 student scores into an array
long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example eC.24 shows how an array is passed to a function. The input argument `arr` is simply the address of the 1st element of an array. Often the number of elements in an array is also passed as an input argument. In a function, an input argument of type `int[]` indicates that it is an array of integers. Arrays of any type may be passed to a function.

C Code Example eC.24 PASSING AN ARRAY AS AN INPUT ARGUMENT

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
```

```
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;

  avg = getMean(data, 4);

  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, `getMean` could also have been declared as

```
float getMean(int *arr, int len);
```

Although functionally equivalent, `datatype[]` is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

A function is limited to a single output, i.e., return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself. C Code Example eC.25 sorts an array from lowest to highest and leaves the result in the same array. The three function prototypes below are equivalent. The length of an array in a function declaration (i.e., `int vals[100]`) is ignored.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

C Code Example eC.25 PASSING AN ARRAY AND ITS LENGTH AS INPUTS

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len)
{
  int i, j, temp;

  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Arrays may have multiple dimensions. C Code Example eC.26 uses a two-dimensional array to store the grades across eight problem sets for ten students. Recall that initialization of array values using `{}` is only allowed at declaration.

C Code Example eC.26 TWO-DIMENSIONAL ARRAY INITIALIZATION

```
// Initialize 2-D array at declaration
 int grades[10][8] = { {100, 107, 99,  101, 100, 104, 109, 117},
                       {103, 101, 94,  101, 102, 106, 105, 110},
                       {101, 102, 92,  101, 100, 107, 109, 110},
                       {114, 106, 95,  101, 100, 102, 102, 100},
                       {98,  105, 97,  101, 103, 104, 109, 109},
                       {105, 103, 99,  101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,  107, 98,  101, 109, 104, 110, 108} };
```

C Code Example eC.27 shows some functions that operate on the 2-D `grades` array from C Code Example eC.26. Multi-dimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

C Code Example eC.27 OPERATING ON MULTI-DIMENSIONAL ARRAYS

```
#include <stdio.h>

// Print the contents of a 10×8 array
void print2dArray(int arr[10][8])
{
  int i, j;

  for (i=0; i<10; i++) {          // for each of the 10 students
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}

// Calculate the mean score of a 10×8 array
float getMean(int arr[10][8])
{
  int i, j;
  float mean, total = 0;

  // get the mean value across a 2D array
  for (i=0; i<10; i++) {
    for (j=0; j<8; j++) {
      total += arr[i][j];       // sum array values
    }
  }
  mean = total/(10*8);
  printf("Mean is: %f\n", mean);
  return mean;
}
```

Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators. Instead, you must use a loop to copy or compare each element one at a time.

C.8.3 Characters

A character (`char`) is an 8-bit variable. It can be viewed either as a two's complement number between −128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. For example, the letter A has the ASCII code 0x41, B=0x42, etc. Thus `'A' + 3` is 0x44, or `'D'`. Table 6.5 lists the ASCII character encodings, and Table eC.4 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (`\r`), newline (`\n`), horizontal tab (`\t`), and the end of a string (`\0`). `\r` is shown for completeness but is rarely used in C programs. `\r` returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. `\n`, instead, moves the location of typing to the beginning of a new line.[2] The *NULL* character (`'\0'`) indicates the end of a text string and is discussed next in Section C.8.4.

The term "carriage return" originates from typewriters that required the carriage, the contraption that holds the paper, to move to the right in order to allow typing to begin at the left side of the page. A carriage return lever, shown on the left in the figure below, is pressed so that the carriage would both move to the right and advance the paper by one line, called a line feed.

A Remington electric typewriter used by Winston Churchill. (*http://cwr.iwm.org.uk/server/show/conMediaFile.71979*)

Table eC.4 Special characters

Special Character	Hexadecimal Encoding	Description
`\r`	0x0D	carriage return
`\n`	0x0A	new line
`\t`	0x09	tab
`\0`	0x00	terminates a string
`\\`	0x5C	backslash
`\"`	0x22	double quote
`\'`	0x27	single quote
`\a`	0x07	bell

---

[2] Windows text files use `\r\n` to represent end-of-line while UNIX-based systems use `\n`, which can cause nasty bugs when moving text files between systems.

C strings are called *null terminated* or *zero terminated* because the length is determined by looking for a zero at the end. In contrast, languages such as Pascal use the first byte to specify the string length, up to a maximum of 255 characters. This byte is called the *prefix byte* and such strings are called *P-strings*. An advantage of null-terminated strings is that the length can be arbitrarily great. An advantage of P-strings is that the length can be determined immediately without having to inspect all of the characters of the string.

C.8.4 Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the null terminator (ASCII value 0x00) at the end of the string.

C Code Example eC.28 shows the declaration of a 10-element character array called `greeting` that holds the string `"Hello!"`. For concreteness, suppose `greeting` starts at memory address 0x50. Figure eC.5 shows the contents of memory from 0x50 to 0x59 holding the string `"Hello!"` Note that the string only uses the first seven elements of the array, even though ten elements are allocated in memory.

C Code Example eC.28 STRING DECLARATION

```
char greeting[10] = "Hello!";
```

C Code Example eC.29 shows an alternate declaration of the string `greeting`. The pointer `greeting` holds the address of the 1st element of a 7-element array comprised of each of the characters in "Hello!" followed by the null terminator. The code also demonstrates how to print strings by using the `%s` format code.

C Code Example eC.29 ALTERNATE STRING DECLARATION

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =. Each element of the character array must be individually copied from the source string to the target string. This is true for any array. C Code Example eC.30 copies one string, `src`, to another, `dst`. The sizes of the arrays are not needed, because the end of the `src` string is indicated by the null terminator. However, `dst` must be large enough so that you don't stomp on other data. `strcpy` and other string manipulation functions are available in C's built-in libraries (see Section C.9.4).

C Code Example eC.30 COPYING STRINGS

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src)
{
  int i = 0;

  do {
    dst[i] = src[i];        // copy characters one byte at a time
  } while (src[i++]);       // until the null terminator is found
}
```

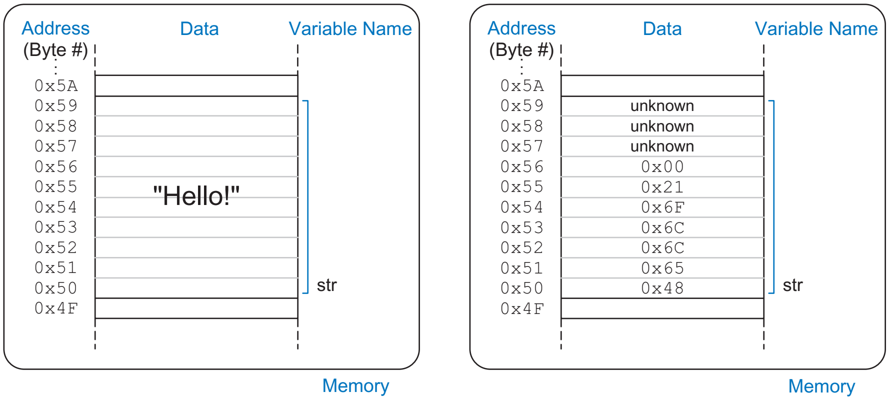

Address
(Byte #)
Data
Variable Name
0x5A
0x59
0x58
0x57
0x56
0x55
0x54
0x53
0x52
0x51
0x50
0x4F
"Hello!"
str
Memory
Address
(Byte #)
Data
Variable Name
0x5A
0x59 unknown
0x58 unknown
0x57 unknown
0x56 0x00
0x55 0x21
0x54 0x6F
0x53 0x6C
0x52 0x6C
0x51 0x65
0x50 0x48
0x4F
str
Memory

Figure eC.5 **The string "Hello!" stored in memory**

C.8.5 Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

where `struct` is a keyword indicating that it is a structure, `name` is the structure tag name, and `element1` and `element2` are members of the structure. A structure may have any number of members. C Code Example eC.31 shows how to use a structure to store contact information. The program then declares a variable `c1` of type `struct contact`.

C Code Example eC.31 STRUCTURE DECLARATION

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
```

```
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures. C Code Example eC.32 creates an array of contacts.

C Code Example eC.32 ARRAY OF STRUCTURES

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure. C Code Example eC.33 shows an example of declaring a pointer to a `struct contact`, assigning it to point to the 42nd element of `classlist` from C Code Example eC.32, and using the member access operator to set a value in that element.

C Code Example eC.33 ACCESSING STRUCTURE MEMBERS USING POINTERS AND ->

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9;  // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference. Passing by value requires the compiler to copy the entire structure into memory for the function to access. This can require a large amount of memory and time for a big structure. Passing by reference involves passing a pointer to the structure, which is more efficient. The function can also modify the structure being pointed to rather than having to return another structure. C Code Example eC.34 shows two versions of the `stretch` function that makes a contact 2 cm taller. `stretchByReference` avoids copying the large structure twice.

C Code Example eC.34 PASSING STRUCTURES BY VALUE OR BY NAME

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
```

```
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}

int main(void)
{
  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);    // and stretch some more
}
```

C.8.6 `typedef`

C also allows you to define your own names for data types using the `typedef` statement. For example, writing `struct contact` becomes tedious when it is often used, so we can define a new type named `contact` and use it as shown in C Code Example eC.35.

C Code Example eC.35 CREATING A CUSTOM TYPE USING `typedef`

```
typedef struct contact {
  char name[30];
  int phone;
  float height; // in meters
} contact;        // defines contact as shorthand for "struct contact"

contact c1;       // now we can declare the variable as type contact
```

`typedef` can be used to create a new type occupying the same amount of memory as a primitive type. C Code Example eC.36 defines `byte` and `bool` as 8-bit types. The `byte` type may make it clearer that the purpose of `pos` is to be an 8-bit number rather than an ASCII character. The `bool` type indicates that the 8-bit number is representing TRUE or FALSE. These types make a program easier to read than if one simply used `char` everywhere.

C Code Example eC.36 `typedef byte` AND `bool`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

C Code Example eC.37 illustrates defining a 3-element `vector` and a 3 × 3 `matrix` type using arrays.

C Code Example eC.37 `typedef vector` AND `matrix`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 Dynamic Memory Allocation*

In all the examples thus far, variables have been declared *statically*; that is, their size is known at compile time. This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The `malloc` function from `stdlib.h` allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example, the following code allocates 10 shorts (10 × 2 = 20 bytes). The `sizeof` operator returns the size of a type or variable in bytes.

```
// dynamically allocate 20 bytes of memory
short *data = malloc(10*sizeof(short));
```

C Code Example eC.38 illustrates dynamic allocation and de-allocation. The program accepts a variable number of inputs, stores them in a dynamically allocated array, and computes their average. The amount of memory necessary depends on the number of elements in the array and the size of each element. For example, if an `int` is a 4-byte variable and 10 elements are needed, 40 bytes are dynamically allocated. The `free` function de-allocates the memory so that it could later be used for other purposes. Failing to de-allocate dynamically allocated data is called a *memory leak* and should be avoided.

C Code Example eC.38 DYNAMIC MEMORY ALLOCATION AND DE-ALLOCATION

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example eC.24.

int main(void) {
  int len, i;
  int *nums;

  printf("How many numbers would you like to enter?    ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:    ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8 Linked Lists*

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*. Linked lists illustrate many of the concepts of structures, pointers, and dynamic memory allocation.

C Code Example eC.39 describes a linked list for storing computer user accounts to accommodate a variable number of users. Each user has a user name, a password, a unique user identification number (UID), and a field indicating whether they have administrator privileges. Each element of the list is of type `userL`, containing all of this user information along with a link to the next element in the list. A pointer to the head of the list is stored in a global variable called `users`, and is initially set to NULL to indicate that there are no users.

The program defines functions to insert, delete, and find a user and to count the number of users. The `insertUser` function allocates space for a new list element and adds it to the head of the list. The `deleteUser` function scans through the list until the specified UID is found and then removes that element, adjusting the link from the previous element to skip the deleted element and freeing the memory occupied by the deleted element. The `findUser` function scans through the list until the specified UID is found and returns a pointer to that element, or NULL if the UID is not found. The `numUsers` function counts the number of elements in the list.

C Code Example eC.39 LINKED LIST

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];      // user name
  char passwd[80];     // password
  int uid;             // user identification number
  int admin;           // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));  // create space for new user
  strcpy(newUser->uname, uname);    // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;            // insert at start of linked list
  users = newUser;
}

void deleteUser(int uid) {  // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;        // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

SUMMARY

- **Pointers:** A pointer holds the address of a variable.
- **Arrays:** An array is a list of similar elements declared using square brackets `[]`.
- **Characters:** `char` types can hold small integers or special codes for representing text or symbols.
- **Strings:** A string is an array of characters ending with the null terminator 0x00.
- **Structures:** A structure stores a collection of related variables.
- **Dynamic memory allocation:** `malloc` is a built-in functions for allocating memory as the program runs. `free` de-allocates the memory after use.
- **Linked Lists:** A linked list is a common data structure for storing a variable number of elements.

C.9 STANDARD LIBRARIES

Programmers commonly use a variety of standard functions, such as printing and trigonometric operations. To save each programmer from having to write these functions from scratch, C provides *libraries* of frequently used functions. Each library has a header file and an associated object file, which is a partially compiled C file. The header file holds variable declarations, defined types, and function prototypes. The object file contains the functions themselves and is linked at compile-time to create the executable. Because the library function calls are already compiled into an object file, compile time is reduced. Table eC.5 lists some of the most frequently used C libraries, and each is described briefly below.

C.9.1 stdio

The standard input/output library `stdio.h` contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

```
#include <stdio.h>
```

`printf`

The *print formatted* statement `printf` displays text to the console. Its required input argument is a string enclosed in quotes " ". The string contains text and optional commands to print variables. Variables to be printed are listed after the string and are printed using format codes shown in Table eC.6. C Code Example eC.40 gives a simple example of `printf`.

Table eC.5 **Frequently used C libraries**

C Library Header File	Description
`stdio.h`	**Standard input/output library**. Includes functions for printing or reading to/from the screen or a file (`printf`, `fprintf` and `scanf`, `fscanf`) and to open and close files (`fopen` and `fclose`).
`stdlib.h`	**Standard library**. Includes functions for random number generation (`rand` and `srand`), for dynamically allocating or freeing memory (`malloc` and `free`), terminating the program early (`exit`), and for conversion between strings and numbers (`atoi`, `atol`, and `atof`).
`math.h`	**Math library**. Includes standard math functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor`, and `ceil`.
`string.h`	**String library**. Includes functions to compare, copy, concatenate, and determine the length of strings.

Table eC.6 `printf` **format codes for printing variables**

Code	Format
`%d`	Decimal
`%u`	Unsigned decimal
`%x`	Hexadecimal
`%o`	Octal
`%f`	Floating point number (`float` or `double`)
`%e`	Floating point number (`float` or `double`) in scientific notation (e.g., 1.56e7)
`%c`	Character (`char`)
`%s`	String (null-terminated array of characters)

C Code Example eC.40 PRINTING TO THE CONSOLE USING `printf`

```
// Simple print function
#include <stdio.h>

int num = 42;
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Console Output:

```
The answer is 42.
```

Floating point formats (`floats` and `doubles`) default to printing six digits after the decimal point. To change the precision, replace `%f` with `%w.df`, where `w` is the minimum width of the number, and `d` is the number of decimal places to print. Note that the decimal point is included in the width count. In C Code Example eC.41, `pi` is printed with a total of four characters, two of which are after the decimal point: 3.14. `e` is printed with a total of eight characters, three of which are after the decimal point. Because it only has one digit before the decimal point, it is padded with three leading spaces to reach the requested width. `c` should be printed with five characters, three of which are after the decimal point. But it is too wide to fit, so the requested width is overridden while retaining the three digits after the decimal point.

C Code Example eC.41 FLOATING POINT NUMBER FORMATS FOR PRINTING

```
// Print floating point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne  = %8.3f\nc  = %5.3f\n", pi, e, c);
```

Console Output:

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Because `%` and `\` are used in print formatting, to print these characters themselves, you must use the special character sequences shown in C Code Example eC.42.

C Code Example eC.42 PRINTING % AND \ USING `printf`

```
// How to print % and \ to the console
printf("Here are some special characters: %% \\ \n");
```

Console Output:

```
Here are some special characters: % \
```

`scanf`

The `scanf` function reads text typed on the keyboard. It uses format codes in the same way as `printf`. C Code Example eC.43 shows how to use `scanf`. When the `scanf` function is encountered, the program waits until the user types a value before continuing execution. The arguments to `scanf` are a string indicating one or more format codes and pointers to the variables where the results should be stored.

C Code Example eC.43 READING USER INPUT FROM THE KEYBOARD WITH `scanf`

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;

  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);    // note no & needed: str is a pointer
}
```

File Manipulation

Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information. In C, the file must first be opened with the `fopen` function. It can then be read or written with `fscanf` or `fprintf` in a way analogous to reading and writing to the console. Finally, it should be closed with the `fclose` command.

The `fopen` function takes as arguments the file name and a *print mode*. It returns a *file pointer* of type `FILE*`. If `fopen` is unable to open the file, it returns NULL. This might happen when one tries to read a nonexistent file or write a file that is already opened by another program. The modes are:

`"w"`: Write to a file. If the file exists, it is overwritten.

`"r"`: Read from a file.

`"a"`: Append to the end of an existing file. If the file doesn't exist, it is created.

C Code Example eC.44 shows how to open, print to, and close a file. It is good practice to always check if the file was opened successfully and to provide an error message if it was not. The `exit` function will be discussed in Section C.9.2. The `fprintf` function is like `printf` but it also takes the file pointer as an input argument to know which file to write. `fclose` closes the file, ensuring that all of the information is actually written to disk and freeing up file system resources.

C Code Example eC.44 PRINTING TO A FILE USING `fprintf`

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1);  // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

It is idiomatic to open a file and check if the file pointer is NULL in a single line of code, as shown in C Code Example eC.44. However, you could just as easily separate the functionality into two lines:

```
fptr = fopen(“result.txt”, “w”);
if (fptr == NULL)
  ...
```

C Code Example eC.45 illustrates reading numbers from a file named data.txt using `fscanf`. The file must first be opened for reading. The program then uses the `feof` function to check if it has reached the end of the file. As long as the program is not at the end, it reads the next number and prints it to the screen. Again, the program closes the file at the end to free up resources.

C Code Example eC.45 READING INPUT FROM A FILE USING `fscanf`

```
#include <stdio.h>

int main(void)
{
  FILE *fptr;
  int data;

  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) {   // check that the end of the file hasn't been reached
    fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }

  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Console Output:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Other Handy stdio Functions

The `sprintf` function prints characters into a string, and `sscanf` reads variables from a string. The `fgetc` function reads a single character from a file, while `fgets` reads a complete line into a string.

`fscanf` is rather limited in its ability to read and parse complex files, so it is often easier to `fgets` one line at a time and then digest that line using `sscanf` or with a loop that inspects characters one at a time using `fgetc`.

C.9.2 stdlib

The standard library `stdlib.h` provides general purpose functions including random number generation (`rand` and `srand`), dynamic memory allocation (`malloc` and `free`, already discussed in Section C.8.7), exiting the program early (`exit`), and number format conversions. To use these functions, add the following line at the top of the C file.

```
#include <stdlib.h>
```

`rand` and `srand`

`rand` returns a pseudo-random integer. Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the *seed*. To convert the number to a particular range, use the modulo operator (`%`) as shown in C Code Example eC.46 for a range of 0 to 9. The values `x` and `y` will be random but they will be the same each time this program runs. Sample console output is given below the code.

C Code Example eC.46 RANDOM NUMBER GENERATION USING `rand`

```
#include <stdlib.h>
int x, y;

x = rand();         // x = a random integer
y = rand() % 10;    // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

A programmer creates a different sequence of random numbers each time a program runs by changing the seed. This is done by calling the `srand` function, which takes the seed as its input argument. As shown in C Code Example eC.47, the seed itself must be random, so a typical C program assigns it by calling the `time` function, that returns the current time in seconds.

For historical reasons, the `time` function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT). This date is just after the UNIX operating system was created by a group at Bell Labs, including Dennis Ritchie and Brian Kernighan, in 1969. Similar to New Year's Eve parties, some UNIX enthusiasts hold parties to celebrate significant values returned by `time`. For example, on February 1, 2009 at 23:31:30 UTC, `time` returned 1,234,567,890. In the year 2038, 32-bit UNIX clocks will overflow into the year 1901.

C Code Example eC.47 SEEDING THE RANDOM NUMBER GENERATOR USING `srand`

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>    // needed to call time()

int main(void)
{
  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

`exit`

The `exit` function terminates a program early. It takes a single argument that is returned to the operating system to indicate the reason for termination. 0 indicates normal completion, while nonzero conveys an error condition.

Format Conversion: `atoi`, `atol`, `atof`

The standard library provides functions for converting ASCII strings to integers, long integers, or doubles using `atoi`, `atol`, and `atof`, respectively, as shown in C Code Example eC.48. This is particularly useful when reading in mixed data (a mix of strings and numbers) from a file or when processing numeric command line arguments, as described in Section C.10.3.

C Code Example eC.48 FORMAT CONVERSION

```
// Convert ASCII strings to ints, longs, and floats
#include <stdlib.h>

int main(void)
{
  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");

  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Console Output:

```
x = 42   y = 833   z = 3.822000
```

C.9.3 math

The math library `math.h` provides commonly used math functions such as trigonometry functions, square root, and logs. C Code Example eC.49 shows how to use some of these functions. To use math functions, place the following line in the C file:

```
#include <math.h>
```

C Code Example eC.49 MATH FUNCTIONS

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);           // 1, note: the input argument is in radians
  b = 2 * acos(0);      // pi (acos means arc cosine)
  c = sqrt(144);        // 12
  d = exp(2);           // e^2 = 7.389056,
  e = log(7.389056);    // 2 (natural logarithm, base e)
  f = log10(1000);     // 3 (log base 10)
  g = floor(178.567);  // 178, rounds to next lowest whole number
  h = pow(2, 10);      // computes 2 raised to the 10th power

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
         a, b, c, d, e, f, g, h);
}
```

Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 string

The string library `string.h` provides commonly used string manipulation functions. Key functions include:

```
// copy src into dst and return dst
char *strcpy(char *dst, char *src);

// concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Although we have introduced relatively simple C programs, real-world programs can consist of tens or even thousands of C files to enable modularity, readability, and multiple programmers. This section describes how to compile a program spread across multiple C files and shows how to use compiler options and command line arguments.

C.10.1 Compiling Multiple C Source Files

Multiple C files are compiled into a single executable by listing all file names on the compile line as shown below. Remember that the group of C files still must contain only one `main` function, conventionally placed in a file named `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2 Compiler Options

Compiler options allow the programmer to specify such things as output file names and formats, optimizations, etc. Compiler options are not

Table eC.7 **Compiler options**

Compiler Option	Description	Example
`-o` outfile	specifies output file name	`gcc -o hello hello.c`
`-S`	create assembly language output file (not executable)	`gcc -S hello.c` this produces `hello.s`
`-v`	verbose mode – prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-Olevel`	specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	`gcc -O3 hello.c`
`--version`	list the version of the compiler	`gcc –version`
`--help`	list all command line options	`gcc --help`
`-Wall`	print all warnings	`gcc -Wall hello.c`

standardized, but Table eC.7 lists ones that are commonly used. Each option is typically preceded by a dash (`-`) on the command line, as shown. For example, the "`-o`" option allows the programmer to specify an output file name other than the `a.out` default. A plethora of options exist; they can be viewed by typing `gcc --help` at the command line.

C.10.3 Command Line Arguments

Like other functions, `main` can also take input variables. However, unlike other functions, these arguments are specified at the command line. As shown in C Code Example eC.50, `argc` stands for *argument count*, and it denotes the number of arguments on the command line. `argv` stands for *argument vector*, and it is an array of the strings found on the command line. For example, suppose the program in C Code Example eC.50 is compiled into an executable called `testargs`. When the lines below are typed at the command line, `argc` has the value 4, and the array `argv` has the values `{"./testargs", "arg1", "25", "lastarg!"}`. Note that the executable name is counted as the 1st argument. The console output after typing this command is shown below C Code Example eC.50.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Programs that need numeric arguments may convert the string arguments to numbers using the functions in `stdlib.h`.

C Code Example eC.50 COMMAND LINE ARGUMENTS

```
// Print command line arguments

#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```

Console Output:

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

C.11 COMMON MISTAKES

As with any programming language, you are almost certain to make errors while you write nontrivial C programs. Below are descriptions of some common mistakes made when programming in C. Some of these errors are particularly troubling because they compile but do not function as the programmer intended.

C Code Mistake eC.1 MISSING & IN `scanf`

Erroneous Code

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // missing & before a
```

Corrected Code:

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

C Code Mistake eC.2 USING = INSTEAD OF == FOR COMPARISON

Erroneous Code

```
if (x = 1)    // always evaluates as TRUE
  printf("Found!\n");
```

Corrected Code

```
if (x == 1)
  printf("Found!\n");
```

Debugging skills are acquired with practice, but here are a few hints.

- Fix bugs starting with the first error indicated by the compiler. Later errors may be downstream effects of this error. After fixing that bug, recompile and repeat until all bugs (at least those caught by the compiler!) are fixed.
- When the compiler says a valid line of code is in error, check the code above it (i.e., for missing semicolons or braces).
- When needed, split up complicated statements into multiple lines.
- Use `printf` to output intermediate results.
- When a result doesn't match expectations, start debugging the code at the *first* place it deviates from expectations.
- Look at all compiler warnings. While some warnings can be ignored, others may alert you to more subtle code errors that will compile but not run as intended.

C Code Mistake eC.3 INDEXING PAST LAST ELEMENT OF ARRAY

Erroneous Code

```
int array[10];
array[10] = 42;    // index is 0-9
```

Corrected Code

```
int array[10];
array[9] = 42;
```

C Code Mistake eC.4 USING = IN `#define` STATEMENT

Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

Corrected Code

```
#define NUM 4
```

C Code Mistake eC.5 USING AN UNINITIALIZED VARIABLE

Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
  ...
```

Corrected Code

```
int i = 10;
if (i == 10)
  ...
```

C Code Mistake eC.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

Erroneous Code

```
#include   "myfile.h"
```

Corrected Code

```
#include   "othercode\myfile.h"
```

C Code Mistake eC.7 USING LOGICAL OPERATORS (`!`, `||`, `&&`) INSTEAD OF BITWISE (`~`, `|`, `&`)

Erroneous Code

```
char x=!5;     // logical NOT: x = 0
char y=5||2;   // logical OR:  y = 1
char z=5&&2;   // logical AND: z = 1
```

Corrected Code

```
char x=~5; // bitwise NOT: x = 0b11111010
char y=5|2;// bitwise OR:  y = 0b00000111
char z=5&2;// logical AND: z = 0b00000000
```

C Code Mistake eC.8 FORGETTING `break` IN A `switch/case` STATEMENT

Erroneous Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default: direction = 0;
}
// direction = 0
```

Corrected Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default: direction = 0;
}
// direction = 2
```

C Code Mistake eC.9 MISSING CURLY BRACES `{}`

Erroneous Code

```
if (ptr == NULL) // missing curly braces
  printf("Unable to open file.\n");
  exit(1);     // always executes
```

Corrected Code

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

C Code Mistake eC.10 USING A FUNCTION BEFORE IT IS DECLARED

Erroneous Code

```
int main(void)
{
  test();
}

void test(void)
{...
}
```

Corrected Code

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

C Code Mistake eC.11 DECLARING A LOCAL AND GLOBAL VARIABLE WITH THE SAME NAME

Erroneous Code

```
int x = 5;    // global declaration of x
int test(void)
{
  int x = 3;  // local declaration of x
  ...
}
```

Corrected Code

```
int x = 5;    // global declaration of x
int test(void)
{
  int y = 3;  // local variable is y
  ...
}
```

C Code Mistake eC.12 TRYING TO INITIALIZE AN ARRAY WITH `{}` AFTER DECLARATION

Erroneous Code

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

Corrected Code

```
int scores[3] = {93, 81, 97};
```

C Code Mistake eC.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

Erroneous Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

scores2 = scores;
```

Corrected Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
  scores2[i] = scores[i];
```

C Code Mistake eC.14 FORGETTING THE SEMI-COLON AFTER A `do/while` LOOP

Erroneous Code

```
int num;
do {
  num = getNum();
} while (num < 100)     // missing ;
```

Corrected Code

```
int num;
do {
  num = getNum();
} while (num < 100);
```

C Code Mistake eC.15 USING COMMAS INSTEAD OF SEMICOLONS IN `for` LOOP

Erroneous Code

```
for (i=0, i < 200, i++)
  ...
```

Corrected Code

```
for (i=0; i < 200; i++)
  ...
```

C Code Mistake eC.16 INTEGER DIVISION INSTEAD OF FLOATING POINT DIVISION

Erroneous Code

```
// integer (truncated) division occurs when
// both arguments of division are integers
float x = 9 / 4;  // x = 2.0
```

Corrected Code

```
// at least one of the arguments of
// division must be a float to
// perform floating point division
float x = 9.0 / 4;  // x = 2.25
```

C Code Mistake eC.17 WRITING TO AN UNINITIALIZED POINTER

Erroneous Code

```
int *y = 77;
```

Corrected Code

```
int x, *y = &x;
*y = 77;
```

C Code Mistake eC.18 GREAT EXPECTATIONS (OR LACK THEREOF)

A common beginner error is to write an entire program (usually with little modularity) and expect it to work perfectly the first time. For non-trivial programs, writing modular code and testing the individual functions along the way are essential. Debugging becomes exponentially harder and more time-consuming with complexity.

Another common error is lacking expectations. When this happens, the programmer can only verify that the code produces a result, not that the result is correct. Testing a program with known inputs and expected results is critical in verifying functionality.

This appendix has focused on using C on a system such as a personal computer. Chapter 9 (available as a web supplement) describes how C is used to program an ARM-based Raspberry Pi computer that can be used in embedded systems. Microcontrollers are usually programmed in C because the language provides nearly as much low-level control of the hardware as assembly language, yet is much more succinct and faster to write.

This page intentionally left blank

Index

Note: Page numbers in *italics* indicate figures, tables and text boxes; page numbers preceded by "e" refer to online material.

0, 8, 22. *See also* LOW, FALSE
1, 8, 22. *See also* HIGH, TRUE
32-bit datapath, 386
32-bit instructions, 329
64-bit architecture, 360
74xx series logic, 533.e1–533.e5
  parts
    2:1 mux (74157), *533.e4*
    3:8 decoder (74138), *533.e4*
    4:1 mux (74153), *533.e4*
    AND (7408), *533.e3*
    AND3 (7411), *533.e3*
    AND4 (7421), *533.e3*
    counter (74161, 74163), *533.e4*
    FLOP (7474), 533.e1, *533.e3*
    NAND (7400), *533.e3*
    NOR (7402), *533.e3*
    NOT (7404), *533.e1*
    OR (7432), *533.e3*
    register (74377), *533.e4*
    tristate buffer (74244), *533.e4*
    XOR (7486), *533.e3*
`#define`, 541.e5–541.e6
`#include`, 541.e6–541.e7. *See also* Standard libraries

A

ABI. *See* Application Binary Interface (ABI)
Abstraction, 4–5
  digital. *See* Digital abstraction
Accumulator, 367
Acorn Computer Group, 296, 472
Acorn RISC Machine, 350
Active low, 74–75
A/D conversion, 531.e31–531.e32
Ad hoc testing, 452
ADCs. *See* Analog-to-digital converters (ADCs)
`ADD`, 297, 536
Adders, 239–246
  carry propagate, 240
  carry-lookahead, 241
  full, 56, 240
  half, 240
  HDL for, 184, 200, 450
  prefix, 243
  ripple-carry, 240
Addition, 14–15, 17–18, 235, 239–246, 297. *See also* Adders
  binary, 14–15
  floating point, 259
  signed binary, 15–17
Address. *See also* Memory
  physical, 509–513
  translation, 509–512
  virtual, 508. *See also* Virtual memory
Addressing modes, ARM, 336
  base, 336
  immediate, 336
  PC-relative, 336
  register, 336
Advanced High-performance Bus (AHB), 531.e54
Advanced Micro Devices (AMD), 296
Advanced microarchitecture, 456–470
  branch prediction. *See* Branch prediction
  deep pipelines. *See* Deep pipelines
  heterogeneous multiprocessors. *See* Heterogeneous multiprocessors
  homogeneous multiprocessors. *See* Homogeneous multiprocessors
  micro-operations. *See* Micro-operations
  multiprocessors. *See* Multiprocessors
  multithreading. *See* Multithreading
  out-of-order processor. *See* Out-of-order processor
  register renaming. *See* Register renaming
  single instruction multiple data. *See* Single instruction multiple data (SIMD)
  superscalar processor. *See* Superscalar processor
Advanced Microcontroller Bus Architecture (AMBA), 531.e54
Advanced RISC Machines, 472
AHB. *See* Advanced High-performance Bus (AHB)
AHB-Lite bus, 531.e54–531.e55
Altera FPGA, 274–279
ALU. *See* Arithmetic/logical unit (ALU)
ALU Decoder, 398–400
ALUControl, 248–250, 392, 395
ALUOp, 398

ALUResult, 392–397
ALUSrc, 396
AMAT. *See* Average memory access time (AMAT)
AMBA. *See* Advanced Microcontroller Bus Architecture (AMBA)
AMD. *See* Advanced Micro Devices (AMD)
AMD64, 368
Amdahl, Gene, 492
Amdahl's Law, 492
American Standard Code for Information Interchange (ASCII), 315–316, 541.e8, 541.e27–541.e28
Analog I/O, 531.e25–531.e32
  A/D conversion, 531.e31–531.e32
  D/A conversion, 531.e25–531.e28
  Pulse-width modulation (PWM), 531.e28–531.e31
Analog-to-digital converters (ADCs), 531.e25, 531.e27, 531.e31–531.e32
Analytical engine, 7–8
AND gate, 20–22, 179
  chips (7408, 7411, 7421), 533.e3
  truth table, 20, 22
  using CMOS transistors, 32–33
`AND`, 303–304
AND-OR (AO) gate, 46
Anode, 27
Antidependence, 464
Application Binary Interface (ABI), 320
Application-specific integrated circuits (ASICs), 533.e9
Architectural state, 338, 364
  for ARM, 385–386
Architecture, 295
  assembly language, 296
    instructions, 297–298
    operands, 298–303
  compiling, assembling, and loading, 339
    assembling, 342–343
    compilation, 340–341
    linking, 343–344
    loading, 344–345
    memory map, 339–340
  evolution of ARM architecture, 350
    64-bit architecture, 360
    digital signal processors (DSPs), 352–356
    floating-point instructions, 357–358
    power-saving and security instructions, 358
    SIMD instructions, 358–360
    Thumb instruction set, 351–352
  machine language, 329
    addressing modes, 336
    branch instructions, 334–335
    data-processing instructions, 329–333
    interpreting, 336–337
    memory instructions, 333–334
    stored program, 337–338
  odds and ends, 345
    exceptions, 347–350
    loading literals, 345–346
    NOP, 346
  programming, 303
    branching, 308–309
    conditional statements, 309–312
    condition flags, 306–308
    function calls, 317–329
    getting loopy, 312–313
    logical and arithmetic instructions, 303–306
    memory, 313–317
  x86 architecture, 360
    big picture, 368
    instruction encoding, 364–367
    instructions, 364
    operands, 362–363
    peculiarities, 367–368
    registers, 362
    status flags, 363–364
Arguments, 317–319, 541.e15
  pass by reference, 541.e22
  pass by value, 541.e22
Arithmetic
  ARM instructions, 303–306
  circuits, 239–255
  C operators, 541.e11–541.e13
  HDL operators, 185
Arithmetic/logical unit (ALU), 248–251, 392
  implementation of, 249
  in processor, 392–430
ARM architecture, evolution of, 296, 350
  64-bit architecture, 360
  digital signal processing (DSP) instructions, 352–356
  floating-point instructions, 357–358
  power-saving and security instructions, 358
  SIMD instructions, 358–360
  Thumb instruction set, 351–352
ARM instructions, 295–369, 535–540
  branch instructions, 308–309, 539
  condition flags, 306–308, 540
  data-processing instructions, 303–306, 535–537
    logical instructions, 303–304
    multiply instructions, 305–306, 537
    shift instructions, 304–305
  formats
    addressing modes, 336
    branch instructions, 334
    data-processing instructions, 329–333
    interpreting, 336–337
    memory instructions, 333–335
    stored program, 337–338
  instruction set, 295
  memory instructions, 301–303, 313–317, 538
  miscellaneous instructions, 345–346, 539
ARM Microcontroller Development Kit (MDK-ARM), *297*
ARM microprocessor, 385
  data memory, 385–388
  instruction memory, 385–388
  multicycle, 406–425
  pipelined, 425–433
  program counter, 385–388
  register file, 385–388
  single-cycle, 390–406, 443–456
  state elements of, 385–388
ARM processors, 470
ARM registers, 299–300
  program counter, 308, 338, 386–387
  register file, 386–387
  register set, 299–300
ARM single-cycle HDL, 443–456
  building blocks, 449–452
  controller, 443
  datapath, 443
  testbench, 452–456
ARM7, 472, *473*
ARM9, *474*
ARM9E, 472
ARMv3 architecture, 472
ARMv4 instruction set, 295, 539
ARMv7 instruction, 472
Arrays, 313–317, 541.e23–541.e29
  accessing, 313–317, 541.e23

  bytes and characters, 315–317, 541.e27–541.e29
  comparison or assignment of, 541.e28
  declaration, 314–317, 541.e23
  indexing, 314–317, 541.e23–541.e27
  initialization, 541.e23–541.e24
  as input argument, 541.e24–541.e25
  multi-dimension, 541.e26–541.e27
ASCII. *See* American Standard Code for Information Interchange (ASCII)
ASICs. *See* Application-specific integrated circuits (ASICs)
`ASR`, 304
Assembler, 339, 541.e44
Assembling, 342–343
Assembly language, ARM, 295–350, 535–540
  instructions, 297–350, 535–540
  operands, 297–303
  translating high-level code to, 339–345
  translating machine language to, 337
Assembly language, x86. *See* x86
Associativity
  in Boolean algebra, 62, 63
  in caches, 493, 498–500
Astable circuits, 119
Asymmetric multiprocessors. *See* Heterogeneous multiprocessors
Asynchronous circuits, 120–123
Asynchronous resettable flip-flops
  definition, 116
  HDL, 194–196
Asynchronous serial link, 531.e17, *531.e17*. *See also* Universal Asynchronous Receiver Transmitter (UART)
AT Attachment (ATA), 531.e61–531.e62
Average memory access time (AMAT), 491, 504

B

`B`, 308–309, 334–336, 396–397
Babbage, Charles, 7
Banked registers, 348–349
Base addressing, 336
Baud rate, 531.e17–531.e19
BCD. *See* Binary coded decimal (BCD)
BCM2835, 531.e3, 531.e4–531.e5, 531.e8, 531.e9, 531.e19
  timer, 531.e23
Behavioral modeling, 173–174
Benchmarks, 389
`BEQ`, 309
Biased exponent, 257
BIC (bit clear), 303–304
big.LITTLE, 469
Big-endian memory, 303
Big-endian order, 178
Binary addition, 14–15. *See also* Adders, Addition
Binary coded decimal (BCD), 258
Binary encoding, 125–126, 129–131
  for divide-by-3 counter, 129–131
  for traffic light FSM, 125–126
Binary numbers
  signed, 15–19
  unsigned, 9–11
Binary to decimal conversion, 10, *10–11*
Binary to hexadecimal conversion, 12
Bipolar junction transistors, 26
Bipolar motor drive, *531.e50*
Bipolar signaling, 531.e18
Bipolar stepper motor, *531.e51*, *531.e52–531.e53*
  AIRPAX LB82773-M1, 531.e51, *531.e51*
  direct drive current, *531.e52*
Bistable element, 109
Bit, 8
  dirty, 506
  least significant, 13, *14*
  most significant, 13, *14*
  sign, 16
  use, 502
  valid, 496
Bit cells, 264–269
  DRAM, 266–267
  ROM, 268–270
  SRAM, 267
Bit swizzling, 188
Bitline, 264
Bitwise operators, 177–179
`BL` (branch and link), 318
Block, 493
Block offset, 500–501
Block size (*b*), 493, 500–501
Blocking and nonblocking assignments, 199–200, 205–209
`BLT`. *See* Branch if less than (`BLT`)
BlueSMiRF silver module, 531.e42–531.e43, *531.e42*
Bluetooth wireless communication, 531.e42–531.e43
  BlueSMiRF silver module, 531.e42–531.e43
  classes, *531.e42*
`BNE`, 310
Boole, George, *8*
Boolean algebra, 60–66
  axioms, 61
  equation simplification, 65–66
  theorems, 61–64
Boolean equations, 58–60
  product-of-sums form, 60
  sum-of-products form, 58–60
Boolean logic, 8. *See also* Boolean algebra, Logic gates
Boolean theorems, 61–64
  associativity, 63
  combining, 62
  commutativity, 63
  complements, 62
  consensus, 62, 64
  covering, 62
  De Morgan's, 63–64
  distributivity, 63
  idempotency, 62
  identity, 62
  involution, 62
  null element, 62
Branch if less than (`BLT`), 334–335
Branch instructions, 308–309
  ARM instructions, 539, *539*
Branch misprediction penalty, 438, 459
Branch prediction, 459–461
Branch target address (BTA), 334–335
Branch target buffer, 459
Branching, 308–309, 334–336
  conditional, *309*
  unconditional, *309*
Breadboards, 533.e18–533.e19
BTA. *See* Branch target address (BTA)
Bubble, 20, 63
  pushing, 63–64, 71–73
Bubble, in pipeline, 435–436
Buffers, 20
  lack of, 117
  tristate, 74–75
Bugs, 175
  in C code, 541.e45–541.e49
Bus, *56*
  tristate, *75*

Bus interfaces, 531.e54–531.e57
  AHB-Lite, 531.e54–531.e55
  memory and peripheral interface example, 531.e55–531.e57
Bypassing, 432. *See also* Forwarding
Byte, 13–14, 315–317. *See also* Characters
  least significant, 13–14
  most significant, 13–14
Byte offset, 495
Byte-addressable memory, 301–302
  big-endian, 302–303
  little-endian, 303

C

C programming, 541.e1–541.e49
  common mistakes. *See* Common mistakes in C
  compiler. *See* Compiler, i_Hlt414277118n C
  conditional statements. *See* Conditional statements
  control-flow statements. *See* Control-flow statements
  data types. *See* Data types
  executing a program, 541.e4
  function calls. *See* Function calls
  loops. *See* Loops
  operators. *See* Operators
  simple program, 541.e3–541.e4
  standard libraries. *See* Standard libraries
  variables. *See* Variables in C
Caches, 489–508
  address fields
    block offset, 500–501
    byte offset, 495
    set bits, 495
    tag, 495
  advanced design, 503–507
  evolution of, in ARM, 507
  multiple level, 504
  organizations, 502
    direct mapped, 494–498
    fully associative, 499–500
    multi-way set associative, 498–499
  parameters
    block, 493
    block size, 493, 500–501
    capacity (*C*), 492–493
    degree of associativity (*N*), 499
    number of sets (*S*), 493
  performance of
    hit, 490–492
    hit rate, 491–492
    miss, 480–492, 505
      capacity, 505
      compulsory, 505
      conflict, 498, 505
      penalty, 500
    miss rate, 491–492
      reducing, 505–506
    miss rate *vs.* cache parameters, *505–506*
  replacement policy, 502–503
  status bits
    dirty bit (*D*), 506
    use bit (*U*), 502
    valid bit (*V*), 496
  write policy, 506–507
    write-back, 506–507
    write-through, 506–507
CAD. *See* Computer-aided design (CAD)
Callee, 317
Callee save rule, 324
Callee-saved registers, 323
Caller save rule, 324
Caller-saved registers, 323
Canonical form. *See* Sum-of-products (SOP) form, Product-of-sums (POS) form
Capacitors, 28
Capacity, of cache, 492–493
Capacity miss, 505
Carry propagate adder (CPA). *See* Carry-lookahead adder (CLA); Prefix adders; Ripple-carry adder
Carry-lookahead adder (CLA), 241–243, *242*
`case` statement, in HDL, 201–203. *See also* Switch/case statement
`casez`, `case?`, in HDL, 205
Cathode, 27
Cathode ray tube (CRT), 531.e36. *See also* VGA (Video Graphics Array) monitor
  horizontal blanking interval, 531.e36
  vertical blanking interval, 531.e36
Character LCDs, 531.e33–531.e36
Characters (`char`), 315–317, 541.e8, 541.e27
  arrays. *See also* Strings
  C type, 541.e27
Chips, 28
  multiprocessors, 468
Chopper constant current drive, 531.e51
Circuits
  74xx series. *See* 74xx series logic
  application-specific integrated (ASICs), 533.e9
  astable, 119
  asynchronous, 120, 122–123
  combinational. *See* Combinational logic
  definition of, 55
  delay, 88–92
  glitches in, 92–95
  multiple-output, 68
  priority, 68
  sequential. *See* Sequential logic
  synchronous, 122–123
  synchronous sequential, 120–123
  synthesized, 176, 179, 181
  timing, 88–95, 141–151
CISC. *See* Complex Instruction Set Computer (CISC) architectures
CLBs. *See* Configurable logic blocks (CLBs)
Clock cycles per instruction (CPI), 390
Clock period, 142, 390
Clock skew, 148–151
Clustered multiprocessors, 470
cmd field, 330, 535, 537
CMOS. *See* Complementary Metal-Oxide-Semiconductor Logic (CMOS)
`CMP`, 402
Combinational composition, 56
Combinational logic, 174
  design, 55–106
    Boolean algebra, 60–66
    Boolean equations, 58–60
    building blocks, 83–88, 239–255
    delays, 88–92
    don't cares, 81–82
    Karnaugh maps (K-maps), 75–83
    multilevel, 66–73
    precedence, 58
    timing, 88–95
    two-level, 69

    - X (contention). *See* Contention (X)
    - X (don't cares). *See* Don't care (X)
    - Z (floating). *See* Floating (Z)
  - HDLs. *See* Hardware description languages (HDLs)
- Combining theorem, 62
- Command line arguments, 541.e44–541.e45
- Comments
  - in ARM assembly, 297–298
  - in C, 297–298, 541.e5
  - in SystemVerilog, 180
  - in VHDL, 180
- Common mistakes in C, 541.e45–541.e49
- Comparators, 246–248
- Comparison
  - in hardware. *See* Comparators; Arithmetic/logical unit (ALU)
  - processor performance, *424–425*
  - using ALU, 251
- Compiler, in C, 339–345, 541.e4–541.e5, 541.e43–541.e44
- Complementary Metal-Oxide-Semiconductor gates (CMOS), 26–34
- Complements theorem, 62
- Complex instruction set computer (CISC) architectures, 298, 361, 458
- Complexity management, 4–7
  - digital abstraction, 4–5
  - discipline, 5–6
  - hierarchy, 6–7
  - modularity, 6–7
  - regularity, 6–7
- Compulsory miss, 505
- Computer-aided design (CAD), 71, 129
- Concurrent signal assignment statement, 179, 183–184, 193, 200–206
- *cond* field, 306–*307*, 330, 535
- Condition flags, 306–308
  - ARM instructions, 540, *540*
- Condition mnemonics, 307
- Conditional assignment, 181–182
- Conditional branches, 308–309
- Conditional Logic, 398–400, 413–415
- Conditional operator, 181–182
- Conditional signal assignments, 181–182
- Conditional statements, 309
  - in ARM assembly
    - `if`, 309–310
    - `if/else`, 310–311
    - `switch/case`, 311–312
  - in C, 541.e17–541.e18
    - `if`, 541.e17–541.e18
    - `if/else`, 541.e17
    - `switch/case`, 541.e17–541.e18
  - in HDL, 194, 201–205
    - `case`, 201–203
    - `casez`, `case?`, 205
    - `if`, `if/else`, 202–205
- Configurable logic blocks (CLBs), 275, 533.e7. *See also* Logic elements (LEs)
- Conflict miss, 505
- Consensus theorem, 62, 64
- Constants
  - in ARM assembly, 300–301. *See also* Immediates
  - in C, 541.e5–541.e6
- Contamination delay, 88–92. *See also* Short path
- Contention (x), 73–74
- Context switching, 467
- Continuous assignment statements, 179, 193, 200, 206
- Control hazard, 432, 437–440
- Control signals, 91, 249
- Control unit, 386. *See also* ALU Decoder, Main Decoder
  - of multicycle ARM processor, 413–423
  - of pipelined ARM processor, 430
  - of single-cycle ARM processor, 397–401
- Control-flow statements
  - conditional statements. *See* Conditional statements
  - loops. *See* Loops
- CoreMark, *389*
- Cortex-A7 and -A15, *475*
- Cortex-A9, *475*
- Counters, 260–261
  - divide-by-3, 130
- Covering theorem, 62
- CPA. *See* Carry propagate adder (CPA)
- CPI. *See* Clock cycles per instruction (CPI); Cycles per instruction (CPI)
- Critical path, 89–92, 402
- Cross-coupled inverters, 109, *110*
  - bistable operation of, *110*
- CRT. *See* Cathode ray tube (CRT)
- Current Program Status Register (CPSR), 306, 324, 347
- Cycle time. *See* Clock period
- Cycles per instruction (CPI), 390, *424*
- Cyclic paths, 120
- Cyclone IV FPGA, 275–279

D

- D flip-flops. *See* Flip-flops
- D latch. *See* La_Hlt414277505tches
- D/A conversion, 531.e25–531.e28
- DACs. *See* Digital-to-analog converters (DACs)
- DAQs. *See* Data Acquisition Systems (DAQs)
- Data Acquisition Systems (DAQs), 531.e62–531.e63
  - myDAQ, 531.e62–531.e63
- Data hazard, 432–436
  - HDL for, 455
- Data memory, 387–388
- Data segment, 340
- Data sheets, 533.e9–533.e14
- Data types, 541.e21–541.e35
  - arrays. *See* Arrays
  - characters. *See* Characters (char)
  - dynamic memory allocation. *See* Dynamic memory allocation (malloc, free)
  - linked list. *See* Linked list
  - pointers. *See* Pointers
  - strings. *See* Strings
  - structures. *See* Structures (struct)
  - typedef, 541.e31–541.e32
- Datapath
  - multicycle ARM processor, 406–413
    - `B` instruction, 412–413
    - `LDR` instruction, 407–410
    - `STR` instruction, 411–412
  - pipelined ARM processor, 428–430
  - single-cycle ARM processor, 390
    - `B` instruction, 396–397

Datapath (*Continued*)
    `LDR` instruction, 391–394
    `STR` instruction, 394–396
Data-processing instructions, *536*
  ARM instructions, 329–333, 396–397, 535–537
  encodings, *536*
DC motors, *531.e43*, 531.e44–531.e48
  H-bridge, 531.e44, *531.e45*
  shaft encoder, 531.e43–531.e44
DC transfer characteristics, 24–26. *See also* Direct current (DC) transfer characteristics, Noise margins
DDR. *See* Double-data rate memory (DDR)
De Morgan, Augustus, 63
De Morgan's theorem, 63–64
DE-9 cable, *531.e19*
Decimal numbers, *9*
Decimal to binary conversion, 11
Decimal to hexadecimal conversion, 13
Decode stage, 425
Decoders
  definition of, 86–87
  HDL for
    behavioral, 202–203
    parameterized, 219
  logic using, 87–88
  Seven-segment. *See* Seven-segment display decoder
Deep pipelines, 457
`Delaymicros` function, *531.e24*
Delays, logic gates. *See also* Propagation delay
  in HDL (simulation only), 188–189
`DeleteUser` function, 541.e33
Dennard, Robert, 266
Destination register (`rd` or `rt`), 393, 409
Device driver, 531.e3, 531.e6–531.e8
Device under test (DUT), 220
Dhrystone, *389*
Dice, 28
Dielectric, 28
Digital abstraction, 4–5, 7–9, 22–26
Digital circuits. *See* Logic
Digital signal processors (DSPs), 352–356, 469
Digital system implementation, 533.e1–533.e35
  74xx series logic. *See* 74xx series logic
  application-specific integrated circuits (ASICs), 533.e9
  assembly of, 533.e17–533.e20
  breadboards, 533.e18–533.e19
  data sheets, 533.e9–533.e14
  economics, 533.e33–533.e35
  logic families, 533.e15–533.e17
  packaging, 533.e17–533.e20
  printed circuit boards, 533.e19–533.e20
  programmable logic, 533.e2–533.e9
Digital-to-analog converters (DACs), 531.e25–531.e28
DIMM. *See* Dual inline memory module (DIMM)
Diodes, 27–28
  p-n junction, *28*
DIPs. *See* Dual-inline packages (DIPs)
Direct current (DC) transfer characteristics, 24, *25*
Direct mapped cache, 494–498, *495*
Direct voltage drive, 531.e51
Dirty bit (*D*), 506
Discipline
  dynamic, 142–151. *See also* Timing analysis
  static, 142–151. *See also* Noise margins
Discrete-valued variables, 7
Distributivity theorem, 63
Divide-by-3 counter
  design of, 129–131
  HDL for, 210–211
Divider, 254–255
Division
  circuits, 254–255
`Do/while` loops, in C, 541.e19–541.e20
Don't care (X), 69, 81–83, 205
Dopant atoms, 27
Double, C type, 541.e8–541.e9
Double-data rate memory (DDR), 268, 531.e60–531.e61
Double-precision formats, 258
DRAM. *See* Dynamic random access memory (DRAM)
DSPs. *See* Digital signal processors (DSPs)
Dual inline memory module (DIMM), 531.e60
Dual-inline packages (DIPs), 28, 533.e1, 533.e17
Dynamic branch predictors, 459
Dynamic data segment, 340
Dynamic discipline, 142–151. *See also* Timing analysis
Dynamic memory allocation (`malloc`, `free`), 541.e32–541.e33
  in ARM memory map, 340
Dynamic power, 34
Dynamic random access memory (DRAM), 266–267, 487–490, 519, 531.e58, 531.e60, 531.e61

E

EasyPIO, 531.e6
Economics, 533.e33
Edge-triggered flip-flop. *See* Flip-flops
EEPROM. *See* Electrically erasable programmable read only memory (EEPROM)
EFLAGS register, 363
Electrically erasable programmable read only memory (EEPROM), 270
Embedded I/O (input/output) systems, 531.e3–531.e32
  analog I/O, 531.e25–531.e32
    A/D conversion, 531.e31–531.e32
    D/A conversion, 531.e25–531.e28
  digital I/O, 531.e8–531.e11
  general-purpose I/O (GPIO), 531.e8–531.e11
  interrupts, 531.e32
  LCDs. *See* Liquid Crystal Displays (LCDs)
  microcontroller peripherals, 531.e32–531.e53
  motors. *See* Motors
  serial I/O, 531.e11–531.e23. *See also* Serial I/O
  timers, 531.e23–531.e24
  VGA monitor. *See* VGA (Video Graphics Array) monitor
Enabled flip-flops, 115–116
Enabled registers, 196–197. *See also* Flip-flops
`EOR` (XOR), 303–304
EPROM. *See* Erasable programmable read only memory (EPROM)

- Equality comparator, 247
- Equation minimization
  - using Boolean algebra, 65–66
  - using Karnaugh maps. *See* Karnaugh maps (K-maps)
- Erasable programmable read only memory (EPROM), 270, 533.e6
- Ethernet, 531.e61
- Exceptions, 346–350
  - banked registers, 348–349
  - exception-related instructions, 349–350
  - exception vector table, 347–348
  - execution modes and privilege levels, 347
  - handler, 340, 349
  - start-up, 350
- Execution time, 389
- `exit`, 541.e41
- Extended instruction pointer (EIP), 362
- *ExtImm*, 408

F

- `factorial` function call, 326
  - stack during, *327*
- Factoring state machines, 134–136
- False, 8, 20, 35, 58, 60, 74, 111, 112, 113, 116, 124, 196
- Fast Fourier Transform (FFT), *352*
- FDIV. *See* Floating-point division (FDIV)
- FFT. *See* Fast Fourier Transform (FFT)
- Field programmable gate arrays (FPGAs), 274–279, 531.e14, 531.e38, 531.e63, 533.e7–533.e9
  - driving VGA cable, *531.e38*
  - in SPI interface, 531.e13–531.e16
- File manipulation, in C, 541.e38–541.e40
- Finite state machines (FSMs), 123–141, 209–213, 413, 417
  - complete multicycle control, 424
  - deriving from circuit, 137–140
  - divide-by-3 FSM, 129–131, 210–211
  - factoring, 134–136, *136*
  - in HDL, 209–213
  - LE configuration for, 277–279
  - Mealy FSM, 132–134
  - Moore FSM, 132–134
  - snail/pattern recognizer FSM, 132–134, 212–213
  - state encodings, 129–131. *See also* Binary encoding, One-cold encoding, One-hot encoding
  - state transition diagram, 124, 125
  - traffic light FSM, 123–129
- Fixed-point numbers, 255–256
- Flags, 250
- Flash memory, 270. *See also* Solid state drive (SSD)
- Flip-flops, 114–118, 193–197. *See also* Registers
  - back-to-back, *145*, 152–157, *197*. *See also* Synchronizers
  - comparison with latches, 118
  - enabled, 115–116
  - HDL for, 451. *See also* Registers
  - metastable state of. *See* Metastability
  - register, 114–115
  - resettable, 116
  - scannable, 262–263
  - shift register, 261–263
  - transistor count, 114, 117
  - transistor-level, 116–117
- `Float`, C type, 541.e6–541.e9
  - print formats of, 541.e36–541.e37
- Floating (Z), 74–75
  - in HDLs, 186–188
- Floating output node, 117
- Floating point division (FDIV) bug, 175
- Floating-gate transistor, 270. *See also* Flash memory
- Floating-point division (FDIV), 259
- Floating-point instructions, ARM, 357–358
- Floating-point numbers, 256–258
  - addition, 259
  - formats, single- and double-precision, 258
  - in programming. *See* Double, C type; Float, C type
  - rounding, 259
  - special cases
    - infinity, 258
    - NaN, 258
- Floating-Point Status and Control Register (FPSCR), 358
- Floating-point unit (FPU), 259
- `For` loops, 312–313, 541.e20
- Format conversion (`atoi`, `atol`, `atof`), 541.e41–541.e42
- Forwarding, 432–435. *See also* Hazards
- FPGAs. *See* Field programmable gate arrays (FPGAs)
- FPU. *See* Floating-point unit (FPU)
- FPSCR. *See* Floating-Point Status and Control Register (FPSCR)
- Frequency shift keying (FSK), 531.e42
  - and GFSK waveforms, *531.e42*
- Front porch, 531.e37
- FSK. *See* Frequency shift keying (FSK)
- FSMs. *See* Finite state machines (FSMs)
- Full adder, 56, 182, 184, 200, 240
  - using always/process statement, 200
- Fully associative cache, 499–500
- *funct* field, 330, 333
- Function calls, 317, 541.e15–541.e16
  - additional arguments and local variables, 328–329
  - arguments, 319, 541.e15
  - leaf, 324–326
  - multiple registers, loading and storing, 322
  - naming conventions, 541.e16
  - with no inputs or outputs, 318, 541.e15
  - nonleaf, 324–326
  - preserved registers, 322–324
  - prototypes, 541.e16
  - recursive, 326–328
  - return, 318–319, 541.e15
  - stack, use of, 320–322. *See also* Stack
- Furber, Steve, *473*
- Fuse-programmable ROM, 269–270

G

- Gates
  - AND, 20, 22, 128
  - buffer, 20
  - multiple-input, 21–22
  - NAND, 21, 31
  - NOR, 21–22, 111, 128
  - NOT, 20
  - OR, 21
  - transistor-level. *See* Transistors
  - XNOR, 21
  - XOR, 21

- General-purpose I/O (GPIO), 531.e8–531.e11
  - switches and LEDs example, 531.e8
- Generate signal, 241, 243
- `Genwaves` function, *531.e27*
- Glitches, *92–95*
- Global data segment, 340
- GPIO. *See* General-purpose I/O (GPIO)
- Graphics accelerators, 469
- Graphics processing units (GPUs), *460*
- Gray, Frank, 76
- Gray codes, 76
- Ground (GND), 22
  - symbol for, 31

H

- Half adder, 240, *240*
- Hard disk, 490–491. *See also* Hard drive
- Hard drive, 490, 508. *See also* Hard disk, Solid state drive (SSD), Virtual memory
- Hardware description languages (HDLs), 443–456. *See also* SystemVerilog, VHSIC Hardware Description Language (VHDL)
  - 2:1 multiplexer, *452*
  - adder, *450*
  - capacity, 505
  - combinational logic, 174, 198
    - bitwise operators, 177–179
    - blocking and nonblocking assignments, 205–209
    - `case` statements, 201–202
    - conditional assignment, 181–182
    - delays, 188–189
  - data memory, *455*
  - data types, 213–217
  - history of, 174–175
  - `if` statements, 202–205
    - internal variables, 182–184
    - numbers, 185
    - operators and precedence, 184–185
    - reduction operators, 180–181
  - immediate extension, *451*
  - instruction memory, 455–456
  - modules, 173–174
  - parameterized modules, 217–220
  - processor building blocks, 449–452
  - register file, *450*
  - resettable flip-flop, *451*
  - resettable flip-flop with enable, *452*
  - sequential logic, 193–198, 209–213
  - simulation and synthesis, 175–177
  - single-cycle ARM processor, 443–456
  - structural modeling, 190–193
  - testbench, 220–224, 452–453
  - top-level module, *454*
- Hardware handshaking, 531.e18
- Hardware reduction, 70–71. *See also* Equation minimization
- Hazard unit, 432–435
- Hazards. *See also* Hazard unit
  - control hazards, 432, 437–440
  - data hazards, 432–436
  - pipelined processor, 431–441
  - read after write (RAW), 431, 464
  - solving
    - control hazards, 437–440
    - forwarding, 432–434
    - stalls, 435–436
  - write after read (WAR), 464
  - write after write (WAW), 465
- H-bridge control, 531.e45
- HDL. *See* Hardware description languages (HDLs), SystemVerilog; VHSIC Hardware Description Language (VHDL)
- Heap, 340
- Heterogeneous multiprocessors, 469–470
- Hexadecimal numbers, 11–13
- Hexadecimal to binary and decimal conversion, 11, *12*
- Hierarchy, 6
- HIGH, 23. *See also* 1, ON
- High-level programming languages, 303, 541.e2
  - compiling, assembling, and loading, 339–345
  - translating into assembly, 300
- High-performance microprocessors, 456
- Hit, 490
- Hit rate, 491
- Hold time constraint, 142–148
  - with clock skew, 149–151
- Hold time violations, 145, *146*, 147–148, 150–151
- Homogeneous multiprocessors, 468–469
- Hopper, Grace, *340*

I

- I/O. *See* Input/output (I/O) systems
- IA-32 architecture. *See* x86
- IA-64, *368*
- ICs. *See* Integrated circuits (ICs)
- Idempotency theorem, 62
- Identity theorem, 62
- Idioms, 177
- `if` statements
  - in ARM assembly, 309–310
  - in C, 541.e17
  - in HDL, 202–205
- `if/else` statements, 310, 541.e27
  - in ARM assembly, 310–311
  - in C, 541.e17–541.e18
  - in HDL, 202–205
- ILP. *See* Instruction level parallelism (ILP)
- IM. *See* Instruction memory
- *imm8* field, 330-331
- *imm12* field, 333
- *imm24* field, 334
- Immediate addressing, 336
- Immediate extension, 451
- Immediates, 300–301, 330–332, 345–346. *See also* Constants
- Implicit leading one, 257
- Information, amount of, 8
- Initializing
  - arrays in C, 541.e23–541.e24
  - variables in C, 541.e11
- Input/Output (I/O) systems, 531.e1–531.e64
  - device driver, 531.e3, 531.e6–531.e8
  - embedded I/O systems. *See* Embedded I/O (input/output) systems
  - I/O registers, 531.e3
  - memory-mapped I/O, 531.e1–531.e3
  - personal computer I/O systems. *See* Personal computer (PC) I/O systems
- Input/output elements (IOEs), 275
- Institute of Electrical and Electronics Engineers (IEEE), 257–258

Instruction encoding, x86, 364–367, *366*
Instruction formats, ARM, 328
  addressing modes, 336
  branch instructions, 334–335
  data-processing instructions, 329–333
  interpreting, 336–337
  memory instructions, 333–335
  stored program, 337–338
Instruction formats, x86, 364–367
Instruction level parallelism (ILP), 465, 467, 468
Instruction memory, 387, 427, *455*
Instruction register (IR), 407, 414
Instruction set, 295
  for ARM, 386
Instruction set. *See also* Architecture
Instructions, x86, 360–368
Instructions, ARM, 295–360, 535–540
  branch instructions, 308–309, 539
  condition flags, 306–308, 540
  data-processing instructions, 535
  logical, 303–304, 536–537
  memory instructions, 301–303, 313–317, 333–334, 538
  miscellaneous instructions, 539
  multiply instructions, 305–306, 537
  shift instructions, 304–305
Instructions per cycle (IPC), 390
Integrated circuits (ICs), *533.e17*
Intel. *See* x86
Intel processors, 360
Intel x86. *See* x86
Interrupts, 347, 531.e32
Invalid logic level, 186
Inverters, 20, *119*, *178*. *See also* NOT gate
  cross-coupled, 109, *110*
  in HDL, 178, 199
An Investigation of the Laws of Thought (Boole), *8*
Involution theorem, 62
IOEs. *See* Input/output elements (IOEs)
IPC. *See* Instructions per cycle (IPC)
IR. *See* Instruction register (IR)
IRWrite, 407, 414

J

Java, 303. *See also* Language

K

Karnaugh, Maurice, 75
Karnaugh maps (K-maps), 75–84, 93–95, 126
  logic minimization using, 77–83
  prime implicants, 65, 77–81, 94–95
  seven-segment display decoder, 79–81
    with “don't cares”, 81–82
Kilobit (Kb/Kbit), 14
Kilobyte (KB), 14
K-maps. *See* Karnaugh maps (K-maps)

L

LAB. *See* Logic array block (LAB)
Land grid array, 531.e58
Language. *See also* Instructions
  assembly, 296–303
  machine, 329–338
  mnemonic, 297
Last-in-first-out (LIFO) queue, 320. *See also* Stack
Latches, 111–113
  comparison with flip-flops, 109, 118
  D, 113, *120*
  SR, 111–113, *112*
  transistor-level, 116–117
Latency, 157–160, 425, 435
Lattice, silicon, 27
LCDs. *See* Liquid crystal displays (LCDs)
`LDR`, 301–303, 313–317, 333–334, 391–394, 538
  critical paths for, *402*
Leaf function, 324
Leakage current, 34
Least recently used (LRU) replacement, 502–503
  two-way associative cache with, 502–503, *503*
Least significant bit (lsb), 13, *14*
Least significant byte (LSB), 13, *14*, 301
LEs. *See* Logic elements (LEs)
Level-sensitive latch. *See* La_Hlt414277542tches: D
LIFO. *See* Last-in-first-out (LIFO) queue
Line options, compiler and command, 341–343, 541.e43–541.e45
Linked list, 541.e33–541.e34
Linker, 340–341
Linking, 339
Linux, 531.e23–531.e24
Liquid crystal displays (LCDs), 531.e33–531.e36
Literal, 58, 96
  loading, 345–346
Little-endian bus order in HDL, 178
Little-endian memory addressing, 303
Load register instruction (`LDR`), 301–302
Loading literals, 345–346
Loads, 344–345
  base addressing of, 336
Local variables, 328–329
Locality, 488
Logic
  bubble pushing, 71–73
  combinational. *See* Combinational logic
  families, 25–26, 533.e15–533.e17, *533.e15*, *533.e17*
  gates. *See* Gates
  hardware reduction, 70–71
  multilevel. *See* Multilevel combinational logic
  programmable, 533.e2–533.e9
  sequential. *See* Sequential logic
  transistor-level. *See* Transistors
  two-level, 69
Logic array block (LAB), 276
Logic arrays, 271–280. *See also* Field programmable gate arrays (FPGAs), Programmable logic arrays (PLAs)
  transistor-level implementation, 279–280
Logic elements (LEs), 275–279
  of Cyclone IV, 276–277
  functions built using, 277–279
Logic families, 25–26, 533.e15–533.e17, *533.e15*, *533.e17*
  compatibility of, 26
  logic levels of, 25
  specifications, *533.e15*, *533.e17*
Logic gates, 19–22, *179*, 533.e2
  AND. *See* AND gate
  AND-OR (AO) gate, 46
  with delays in HDL, *189*
  multiple-input gates, 21–22
  NAND. *See* NAND gate
  NOR. *See* NOR gate

- Logic gates (*Continued*)
  - NOT. *See* NOT gate
  - OR. *See* OR gate
  - OR-AND-INVERT (OAI) gate, 46
  - XNOR. *See* XNOR gate
  - XOR. *See* XOR gate
- Logic levels, 22–26
- Logic simulation, 175–176
- Logic synthesis, 176–177, *176*
- Logical instructions, 303–304
- Logical shifter, 251
- Lookup tables (LUTs), 270, 275–276
- Loops, 312–313, 541.e19–541.e20
  - in ARM assembly
    - `for`, 312–313
    - `while`, 312
  - in C
    - `do/while`, 541.e19–541.e20
    - `for`, 541.e20
    - `while`, 541.e19
- Lovelace, Ada, 338
- LOW, 23. *See also* 0, FALSE
- Low Voltage CMOS Logic (LVCMOS), *25*
- Low Voltage TTL Logic (LVTTL), *25*
- lsb. *See* Least significant bit (lsb)
- LSB. *See* Least significant byte (LSB)
- `LSL`, 304
- `LSR`, 304
- LUTs. *See* Lookup tables (LUTs)
- LVCMOS. *See* Low Voltage CMOS Logic (LVCMOS)
- LVTTL. *See* Low Voltage TTL Logic (LVTTL)

M

- MAC. *See* Multiply-accumulate (MAC)
- Machine code. *See* Machine language
- Machine language, 329
  - addressing modes, 336
  - branch instructions, 334–335
  - data-processing instructions, 329–333
  - interpreting, 336–337
  - memory instructions, 333–335
  - stored program, 337–338, *338*
  - translating to assembly language, *337*
- Magnitude comparator, 247
- Main Decoder, 398–400, *400*
- Main FSM, 413–423, *423*
- `main` function in C, 541.e3
- Main memory, 489–491
- `malloc` function, 541.e32
- Mantissa, 257
- Master-slave flip-flop. *See* Flip-flops
- Masuoka, Fujio, 270
- `math.h`, C library, 541.e42–541.e43
- Max-delay constraint. *See* Setup time constraint
- Maxterms, 58
- MCUs. *See* Microcontroller units (MCUs)
- Mealy machines, 123, *123*, 132–134
  - state transition and output table, *134*
  - state transition diagrams, *133*
  - timing diagrams for, *135*
- Mean time between failure (MTBF), 153–154
- Medium-scale integration (MSI) chips, 533.e2
- *MemWrite*, 394, *397*
- Memory, 313. *See also* Memory arrays
  - access time, 491
  - addressing modes, 363
  - area and delay, 267–268
  - big-endian, 302
  - byte-addressable, 301–303
  - bytes and characters, 315–317
  - HDL for, *272*, *273*, *455–456*
  - hierarchy, *490*
  - little-endian, 303
  - logic using, 270–271
  - main, 490
  - operands in, 301–303
  - physical, *509*
  - ports, 265–266
  - protection, *515*. *See also* Virtual memory
  - types, 266–270
    - DDR, 268
    - DRAM, 266–267
    - flash, 270
    - register file, 268
    - ROM, 268–270
    - SRAM, 266
  - virtual, 490. *See also* Virtual memory
- Memory address computation, *419*
  - data flow during, *419*
- Memory and peripheral interface, 531.e55–531.e57
- Memory arrays, 264–271. *See also* Memory
  - bit cell, 264–270
  - HDL for, *272*, *273*, *455–456*
  - logic using, 270–271
  - organization, 264–265
- Memory hierarchy, 490–491
- Memory instructions, 301–303, 313–317, 333–334, 391–394
  - encodings, 333–334, *538*
- Memory interface, 487–488
- Memory map, ARM, 339–340, 531.e2
- Memory performance. *See* Average Memory Access Time (AMAT)
- Memory protection, *515*
- Memory systems, 487
  - ARM, 507–508
  - performance analysis, 491–492
  - x86, 531.e3
- Memory-mapped I/O, 531.e1–531.e3, *531.e7*
  - address decoder, 531.e1, *531.e2*
  - communicating with I/O devices, *531.e2*
  - hardware, 531.e2, *531.e2*, 531.e3
- *MemtoReg*, 396, *397*
- Metal-oxide-semiconductor field effect transistors (MOSFETs), 26
  - switch models of, 30
- Metastability, 151–157
  - metastable state, 110, 151
  - resolution time, 151–152, 154–157
  - synchronizers, 152–154
- Microarchitecture, 296, 385, 388–389. *See also* Architecture
  - advanced. *See* Advanced microarchitecture
  - architectural state. *See* Architectural state
  - description of, 385–389
  - design process, 386–388
  - evolution of, 470–476
  - HDL representation, 443–456
    - generic building blocks, 449–452
    - single-cycle processor, 444–449
    - testbench, 452–456
  - multicycle processor. *See* Multicycle ARM processor

- performance analysis, 389–390. *See also* Performance analysis
- pipelined processor. *See* Pipelined ARM processor
- real-world perspective, 470–476
- single-cycle processor. *See* Single-cycle ARM processor

Microcontroller, 531.e3, 531.e25
Microcontroller peripherals, 531.e32–531.e53
- Bluetooth wireless communication, 531.e42–531.e43
- character LCD, 531.e33–531.e36
  - control, *531.e35–531.e36*
  - parallel interface, 531.e33
- motor control, 531.e43–531.e53
- VGA monitor, 531.e36–531.e42

Microcontroller units (MCUs), 531.e3
Micro-operations (micro-ops), 458–459
- designers, 456
- high-performance, 456

Microprocessors, 3, *13*, 295
- architectural state of, 338

Millions of instructions per second, 425
Min-delay constraint. *See* Hold time constraint
Minterms, 58
Miss, 490–492, 505
- capacity, 505
- compulsory, 505
- conflict, 498, 505

Miss penalty, 500
Miss rate, 491–492
- and access times, *492*

Misses
- cache, 490
  - capacity, 505
  - compulsory, 505
  - conflict, 505
- page fault, 509–510

`ModR/M` byte, 366
Modularity, 6
Modules, in HDL
- behavioral and structural, 173–174
- parameterized modules, 217–220

Moore, Gordon, *30*
Moore machines, 123, 132
- state transition and output table, 134
- state transition diagrams, *133*
- timing diagrams for, *135*

Moore's law, *30*
MOS transistors. *See* Metal-oxide-semiconductor field effect transistors (MOSFETs)
MOSFET. *See* Metal-oxide-semiconductor field effect transistors (MOSFETs)
Most significant bit (msb), 13, *14*
Most significant byte (MSB), 13, *14*, 301, 302
Motors
- DC, 531.e43, 531.e44–531.e47
- H-bridge, 531.e45–531.e46, *531.e45*, *531.e46*
- servo, 531.e44, 531.e48–531.e49
- stepper, 531.e44, 531.e49–531.e53

`MOV`, 301
MPSSE. *See* Multi-Protocol Synchronous Serial Engine (MPSSE)
msb. *See* Most significant bit (msb)
MSB. *See* Most significant byte (MSB)
MSI chips. *See* Medium-scale integration (MSI) chips
MTBF. *See* Mean time between failure (MTBF)
Multicycle ARM processor, 406
- control, 413–421
- datapath, 407–413
  - `B` instruction, 412–413
  - data-processing instructions, 412
  - `LDR` instruction, 407–410
  - `STR` instruction, 411–412
- performance, 421–425

Multicycle microarchitectures, 388
Multilevel combinational logic, 69–73. *See also* Logic
Multilevel page tables, 516–518
Multiple-output circuit, 68–69
Multiplexers, 83–86
- definition of, 83–84
- HDL for
  - behavioral model of, 181–183
  - parameterized N-bit, 218–219
  - structural model of, 190–193
- logic using, 84–86
- symbol and truth table, 83

Multiplicand, 252–253
Multiplication. *See* Multiplier
Multiplier, 252–253
- HDL for, 253

Multiply instructions, 305–306, 537, *537*
Multiply and multiply-accumulate instructions, *355–356*
Multiply-accumulate (MAC), 352, *356*
Multiprocessors, 468–470
- chip, 468
- heterogeneous, 469–470
- homogeneous, 468

Multi-Protocol Synchronous Serial Engine (MPSSE), *531.e63*
Multithreaded processor, 467
Multithreading, 467–468
Mux. *See* Multiplexers
myDAQ, 531.e62–531.e63

N

NAND (7400), *533.e3*
NAND gate, 21
- CMOS, *31–32*

Nested `if/else` statement, 311, 541.e18
Newton computer, 472
Nibbles, 13–14
nMOS transistors, 28–31, *29–30*
Noise margins, 23–26, *23*
- calculating, 23–24

Nonarchitectural state, 386, 388
Nonblocking and blocking assignments, 199–200, 205–209
Nonleaf function calls, 324–326
Nonpreserved registers, 322–323, 326
`NOP`, 346, 431
NOR gate, 21–22, *63*, *533.e3*
- chip (7402), *533.e3*
- CMOS, 32
- pseudo-nMOS logic, *33*
- truth table, *22*

Not a number (NaN), 258
NOT gate, 20
- chip (7404), *533.e3*
- CMOS, 31

Noyce, Robert, *26*
Null element theorem, 62
Number conversion
- binary to decimal, *10–11*
- binary to hexadecimal, 12
- decimal to binary, 11, 13
- decimal to hexadecimal, 13
- hexadecimal to binary and decimal, 11, *12*
- taking the two's complement, 16

Number systems, 9–19
  binary, 9–11, *10–11*
  comparison of, 18–19, *19*
  estimating powers of two, 14
  fixed-point, 255, *255–256*
  floating-point, 256–259
    addition, 259, 260
    special cases, 258
  hexadecimal, 11–13, *12*
  negative and positive, 15
  sign/magnitude, 15–16
  signed, 15–18
  two's complement, 16–18
  unsigned, 9–11

O

Odds and ends, 345
  exceptions, 346–350
  loading literals, 345–346
  NOP, 346
OFF, 26, *30*
Offset, 302, 392, 408
Offset indexing, ARM, 314
ON, 26, *30*
One-bit dynamic branch predictor, 460
One-cold encoding, 130
One-hot encoding, 129–131
One-time programmable (OTP), 533.e2
op field, 330
Opcode. *See op* field
Operands
  ARM, 298
    constants/immediates, 300–301
    memory, 301–303
    registers, 299
    register set, 300
  x86, 362–363, *363*
Operation code. *See op* field
Operators
  in C, 541.e11–541.e14
  in HDL, 177–185
    bitwise, 177–181
    precedence, 185
    reduction, 180–181
    table of, 185
    ternary, 181–182
OR gate, 21
OR-AND-INVERT (OAI) gate, 46
`ORR` (OR), 303–304
OTP. *See* One-time programmable (OTP)
Out-of-order execution, *466*
Out-of-order processor, 463–465
Output dependence, 465
Overflow
  with addition, 15
  detection, 250–251
Oxide, 28

P

Packages, chips, 533.e17–533.e18
Page fault, 509
Page number, 511
Page offset, 511
Page table, 510–513
Pages, 509
Paging, 516
Parallel I/O, 531.e11
Parallelism, 157–160
Parity gate. *See* XOR gate
Partial products, 252
Pass by reference, 541.e22
Pass by value, 541.e22
Pass gate. *See* Transmission gates
PC. *See* Program counter (PC)
PC Logic, 400
PCB. *See* Printed circuit boards (PCBs)
PCI. *See* Peripheral Component Interconnect (PCI)
PCI express (PCIe), 531.e60
PC-relative addressing, *335*, 336
PCSrc, 394, *395–396*, 440
PCWrite, 410
Perfect induction, proving theorems using, 64–65
Performance analysis, 389–390
  multicycle ARM processor, 422–424
  pipelined ARM processor, 425–428
  processor comparison, 424
  single-cycle ARM processor, 402
Performance Analysis, 389–390. *See also* Average Memory Access Time (AMAT)
Peripheral Component Interconnect (PCI), 531.e59–531.e60
Peripherals devices. *See* Input/output (I/O) systems
Personal computer (PC) I/O systems, 531.e57–531.e64
  data acquisition systems, 531.e62–531.e63
  DDR3 memory, 531.e60–531.e61
  networking, 531.e61
  PCI, 531.e59–531.e60
  SATA, 531.e61–531.e62
  USB, 531.e59, 531.e63–531.e64
Phase locked loop (PLL), *531.e39*
Physical memory, 509
Physical page number (PPN), 511
Physical pages, 509
Pipelined ARM processor, 425–428
  abstract view of, 427
  control unit, 430
  datapath, 428–429
  description, 425–428
  hazards, 431–441
  performance analysis, 441–443
  throughput, 426
Pipelined microarchitecture. *See* Pipelined ARM processor
Pipelining, 158–160
PLAs. *See* Programmable logic arrays (PLAs)
Plastic leaded chip carriers (PLCCs), 533.e17
Platters, 508
PLCCs. *See* Plastic leaded chip carriers (PLCCs)
PLDs. *See* Programmable logic devices (PLDs)
PLL. *See* Phase locked loop (PLL)
pMOS transistors, 28–31, 29
Pointers, 541.e21–541.e23, 541.e25, 541.e28, 541.e30, 541.e32
POS. *See* Product-of-sums (POS) form
Positive edge-triggered flip-flop, 114
Post-indexed addressing, ARM, 314
Power consumption, 34–35
Power-saving and security instructions, 358
PPN. *See* Physical page number (PPN)
Prefix adders, 243–245, *244*
Prefix tree, 245
Pre-indexed addressing, ARM, 314
Preserved registers, 322–324, *323*
Prime implicants, 65, 77
Printed circuit boards (PCBs), 533.e19–533.e20
`printf`, 541.e35–541.e37

Priority
  circuit, 68–69
  encoder, 102–103, 105
Procedure calls. *See* Function calls
Processor performance comparison, 442
  multicycle ARM processor, 424
  pipelined ARM processor, 442
  single-cycle processor, 405
Processor-memory gap, 489
Product-of-sums (POS) form, 60
Program counter (PC), 308, 338, 387, 394
Programmable logic arrays (PLAs), 67, 272–274, 533.e6–533.e7
  transistor-level implementation, 280
Programmable logic devices (PLDs), 533.e6
Programmable read only memories (PROMs), 269, *271*, 533.e2–533.e6
Programming
  in ARM, 303
  arrays. *See* Arrays
  branching. *See* Branching
  in C. *See* C programming
  conditional statements, 309–312
  condition flags, 306–308
  constants. *See* Constants; Immediates
  function calls. *See* Function calls
  getting loopy, 312–313
  logical and arithmetic instructions, 303–306
  loops. *See* Loops
  memory, 313–317
  shift instructions, 304–305
PROMs. *See* Programmable read only memories (PROMs)
Propagate signal, 241
Propagation delay, 88–92. *See also* Critical path
Pseudoinstructions, *346*
Pseudo-nMOS logic, 33–34, *33*
  NOR gate, *33*
  ROMs and PLAs, 279–280
Pulse-Width Modulation (PWM), 531.e28–531.e31
  analog output with, *531.e30–531.e31*
  duty cycle, 531.e28
  signal, *531.e28*
PWM. *See* Pulse-Width Modulation (PWM)

Q

Quiescent supply current, 34

R

Race conditions, 119–120, *120*
`rand`, 541.e40–541.e41
Random access memory (RAM), 266–268, 271, *272*
Raspberry Pi, 531.e3–531.e4, 531.e5, 531.e6, 531.e32, *531.e48–531.e49*
RAW hazard. *See* Read after write (RAW) hazard
Rd field, 330
Read after write (RAW) hazard, 431, 464. *See also* Hazards
Read only memory (ROM), 266, 268–270
  transistor-level implementation, 279–280
Read/write head, 508
ReadData bus, 393, 394
Receiver gate, 22
Recursive function calls, 326–328
Reduced instruction set computer (RISC) architecture, 298, 458
Reduction operators, 180–181
Register file (RF)
  ARM register descriptions, 299
  HDL for, *449*
  in pipelined ARM processor (write on falling edge), 428
  schematic, 268
  use in ARM processor, 387
Register renaming, 465–467
Register set, 300. *See also* Register file (RF)
Registers. *See* ARM registers; Flip-flops; x86 registers
  loading and storing, 322
  preserved and nonpreserved, 322–324
RegSrc, 402
Regularity, 6
RegWrite, 393, 433
Replacement policies, 516
Resettable flip-flops, 116
Resettable registers, 194–196
Resolution time, 151–152. *See also* Metastability
  derivation of, 154–157
Return value, 317
RF. *See* Register file (RF)
Ring oscillator, 119, *119*
Ripple-carry adder, 240, 240–241, 243
RISC architecture. *See* Reduced instruction set computer (RISC) architecture
Rising edge, 88
Rm field, 330
Rn field, 330
ROM. *See* Read only memory (ROM)
`ROR`, 304
rot field, 330-331
Rotations per minute (RPM), 531.e44
Rotators, 251–252
Rounding modes, 259
RPM. *See* Rotations per minute (RPM)
RS-232, 531.e18

S

Sampling, 141
Sampling rate, 531.e25
SATA. *See* Serial ATA (SATA)
Saturated arithmetic, 353
Scalar processor, 461–463, *460*
Scan chains, 262–263
`scanf`, 541.e38
Scannable flip-flop, 262–263
Schematics, rules of drawing, 31, 67
SCK. *See* Serial Clock (SCK)
SDI. *See* Serial Data In (SDI)
SDO. *See* Serial Data Out (SDO)
SDRAM. *See* Synchronous dynamic random access memory (SDRAM)
Segment descriptor, 367
Segmentation, 367
Selected signal assignment statements, 182
Semiconductors, 27
  industry, sales, 3
Sequencing overhead, 143–144, 149, 160, 442
Sequential building blocks. *See* Sequential logic

- Sequential logic, 109–161, 259–263
  - counters, 260
  - finite state machines. *See* Finite state machines (FSMs)
  - flip-flops, 114–118. *See also* Registers
  - latches, 111–113
    - D, 113
    - SR, 111–113
  - registers. *See* Registers
  - shift registers, 261–263
  - timing of. *See* Timing analysis
- Serial ATA (SATA), 531.e62
- Serial Clock (SCK), 531.e12
- Serial communication, with PC, 531.e20
- Serial Data In (SDI), 531.e12
- Serial Data Out (SDO), 531.e12
- Serial I/O, 531.e11–531.e23
  - SPI. *See* Serial peripheral interface (SPI)
  - UART. *See* Universal Asynchronous Receiver Transmitter (UART)
- Serial Peripheral Interface (SPI), 531.e11, 531.e12–531.e17
  - connection between PI and FPGA, *531.e14*
  - ports
    - Serial Clock (SCK), 531.e12
    - Serial Data In (SDI), 531.e12
    - Serial Data Out (SDO), 531.e12
  - register fields in, *531.e13*
  - slave circuitry and timing, *531.e15*
  - waveforms, *531.e12*
- Servo motor, 531.e44, 531.e48–531.e49
- Set bits, 495
- Setup time constraint, 142, 145–147
  - with clock skew, 148–150
- Seven-segment display decoder, 79–82
  - with don't cares, 82–83
  - HDL for, 201–202
- Shaft encoder, 531.e43, 531.e47–531.e48, *531.e48*
- Shift instructions, 304–305, *305*
- Shift registers, 261–263
- Shifters, 251–252
- Short path, 89–92
- Sign bit, 16
- Sign extension, 18
- Sign/magnitude numbers, 15–16, 256
- Signed binary numbers, 15–19
- Signed multiplier, 217
- Silicon dioxide (SiO_2), 28
- Silicon lattice, 27
- SIMD. *See* Single instruction multiple data (SIMD)
- SIMD instructions, 358–360
- `simple` function, 318
- Simple programmable logic devices (SPLDs), 274
- Simulation waveforms, *176*
  - with delays, *189*
- Single instruction multiple data (SIMD), *460*, 472
- Single-cycle ARM processor, 390, *444*
  - Conditional Logic, *447–448*
  - control, 397–401
  - controller, *445*
  - datapath, 390, *448–449*
    - `B` instruction, 396–397
    - data-processing instructions, 395–396
    - `LDR` instruction, 391–394
    - `STR` instruction, 394–396
  - Decoder, *446*
  - instructions, 402
  - performance, 402–405
- Single-cycle microarchitecture, 388
- Single-precision formats, 258. *See also* Floating-point numbers
- Skew. *See* Clock skew
- Slash notation, *56*
- Slave latch, 114. *See also* Flip-flops
- Small-scale integration (SSI) chips, 533.e2
- Solid state drive (SSD), 490. *See also* Flash memory, Hard drive
- SOP. *See* Sum-of-products (SOP) form
- Spatial locality, 488, 500–502
- Spatial parallelism, 157–158
- SPEC, *389*
- SPECINT2000, 424
- SPI. *See* Serial Peripheral Interface (SPI)
- Squashing, 465
- SR latches, 111–113, *112*
- SRAM. *See* Static random access memory (SRAM)
- `srand`, 541.e40–541.e41
- *Src2* field, 330, 333
- SSI chips. *See* Small-scale integration (SSI) chips
- Stack, 320–329. *See also* Function calls
  - during recursive function call, 326–328
  - preserved registers, 322–324
  - stack frame, 322, 328
  - stack pointer (SP), 320
  - storing additional arguments on, 328–329
  - storing local variables on, 328–329
- Stalls, 435–436. *See also* Hazards
- Standard libraries, 541.e35–541.e43
  - math, 541.e42–541.e43
  - stdio, 541.e35–541.e40
    - file manipulation, 541.e38–541.e40
    - `printf`, 541.e35–541.e37
    - `scanf`, 541.e38
  - stdlib, 541.e40–541.e42
    - `exit`, 541.e41
    - format conversion (`atoi`, `atol`, `atof`), 541.e41–541.e42
    - `rand`, `srand`, 541.e40–541.e41
  - string, 541.e43
- State encodings, FSM, 129–131, *134*. *See also* Binary encoding, One-cold encoding, One-hot encoding
- State machine circuit. *See* Finite state machines (FSMs)
- State variables, 109
- Static branch prediction, 459
- Static discipline, 24–26
- Static power, 34
- Static random access memory (SRAM), 266, 267, 519
- Status flags, 363. *See also* Condition flags
- `stdio.h`, C library, 541.e35–541.e40. *See also* Standard libraries
- `stdlib.h`, C library, 541.e40–541.e42. *See also* Standard libraries
- Stepper motors, 531.e44, 531.e49–531.e53
  - bipolar stepper motor, 531.e49, 531.e50–531.e52
    - half-step drive, *531.e50*, 531.e51
    - two-phase-on drive, 531.e50, 531.e51
    - wave drive, *531.e52–531.e53*
- Stored program, 337–338
- `STR`, 394–396
- `string.h`, C library, 541.e43
- Strings, 316–317, 541.e28–541.e29. *See also* Characters (char)
- Structural modeling, 173–174, 190–193
- Structures (`struct`), 541.e29–541.e31

`SUB`, 297
Substrate, 28–29
Subtraction, 17, 246, 297
Subtractor, 246–247
Sum-of-products (SOP) form, 58–60
Superscalar processor, 461–463
Supervisor call (`SVC`) instruction, 349
Supply voltage, 22. *See also* V_{DD}
`SVC`. *See* Supervisor call (`SVC`) instruction
Swap space, 516
`switch/case` statements
  in ARM assembly, 311–312
  in C, 541.e17–541.e18
  in HDL. *See* `case` statement, in HDL
Symbol table, 342, 343
Symmetric multiprocessing (SMP), 468. *See also* Homogeneous multiprocessors
Synchronizers, 152–154, *152–153*
Synchronous circuits, 122–123
Synchronous dynamic random access memory (SDRAM), 268
  DDR, 268
Synchronous logic, design, 119–123
Synchronous resettable flip-flops, 116
Synchronous sequential circuits, 120–123, *122*. *See also* Finite state machines (FSMs)
  timing specification. *See* Timing analysis
SystemVerilog, 173–225. *See also* Hardware description languages (HDLs)
  accessing parts of busses, 188, 192
  bad synchronizer with blocking assignments, 209
  bit swizzling, 188
  blocking and nonblocking assignment, 199–200, 205–208
  `case` statements, 201–202, 205
  combinational logic using, 177–193, 198–208, 217–220
  comments, 180
  conditional assignment, 181–182
  data types, 213–217
  decoders, 202–203, 219
  delays (in simulation), 189
  divide-by-3 FSM, 210–211
  finite state machines (FSMs), 209–213
    Mealy FSM, 213
    Moore FSM, 210, 212
  full adder, 184
    using always/process, 200
    using nonblocking assignments, 208
  history of, 175
  `if` statements, 202–205
  internal signals, 182–184
  inverters, 178, 199
  latches, 198
  logic gates, 177–179
  multiplexers, 181–183, 190–193, 218–219
  multiplier, 217
  numbers, 185–186
  operators, *185*
  parameterized modules, 217–220
    N:2^N decoder, 219
    N-bit multiplexers, 218–219
    N-input AND gate, 220
  priority circuit, 204
    using don't cares, 205
  reduction operators, 180–181
  registers, 193–197
    enabled, 196
    resettable, 194–196
  sequential logic using, 193–198, 209–213
  seven-segment display decoder, 201
  simulation and synthesis, 175–177
  structural models, 190–193
  synchronizer, 197
  testbench, 220–224
    self-checking, 222
    simple, 221
    with test vector file, 223–224
  tristate buffer, 187
  truth tables with undefined and floating inputs, 187, *188*
  z's and x's, 186–188, 205

T

Tag, 495
Taking the two's complement, 16–17
Temporal locality, 488, 493–494, 497, 502
Temporal parallelism, 158–159
Temporary registers, 299
Ternary operators, 181, 541.e13
Testbench, 452–456
Testbenches, HDLs, 220–224
  self-checking, 221–222
  simple, 220–221
  with testvectors, 222–224
Text Segment, 340, 344
Thin small outline package (TSOP), 533.e17
Thread level parallelism (TLP), 467
Threshold voltage, 29
Throughput, 157–160, 388, 425, 468
Thumb instruction set, 351–352
Timers, 531.e23–531.e24
Timing
  of combinational logic, 88–95
    delay. *See* Contamination delay; Propagation delay
    glitches. *See* Glitches
  of sequential logic, 141–157
    analysis. *See* Timing analysis
    clock skew. *See* Clock skew
    dynamic discipline, 141–142
    metastability. *See* Metastability
    resolution time. *See* Resolution time
    system timing. *See* Timing analysis
Timing analysis, 141–151
  calculating cycle time. *See* Setup time constraint
  with clock skew. *See* Clock skew
  hold time constraint. *See* Hold time constraint
  max-delay constraint. *See* Setup time constraint
  min-delay constraint. *See* Hold time constraint
  multicycle processor, 424
  pipelined processor, 441
  setup time constraint. *See* Setup time constraint
  single-cycle processor, 405
TLB. *See* Translation lookaside buffer (TLB)
TLP. *See* Thread level parallelism (TLP)
Transistors, 26–34
  bipolar, 26
  CMOS, 26–33
  gates made from, 31–34
  latches and flip-flops, 116–117
  MOSFETs, 26

Transistors (*Continued*)
  nMOS, 28–34, *29–33*
  pMOS, 28–34, *29–33*
    pseudo-nMOS, 33–34
    ROMs and PLAs, 279–280
    transmission gate, 33
Transistor-Transistor Logic (TTL), 25–26, 533.e15–533.e16
Translating and starting a program, *339*
Translation lookaside buffer (TLB), 514–515
Transmission Control Protocol and Internet Protocol (TCP/IP), 531.e61
Transmission gates, 33
Transmission lines, 533.e20–533.e33
  characteristic impedance (Z_0), 533.e30–533.e31
    derivation of, 533.e30–533.e31
  matched termination, 533.e22–533.e24
  mismatched termination, 533.e25–533.e28
  open termination, 533.e24–533.e25
  reflection coefficient (k_r), 533.e31–533.e32
    derivation of, 533.e31–533.e32
  series and parallel terminations, 533.e28–533.e30
  short termination, 533.e25
  when to use, 533.e28
Transparent latch. *See* Latches: D
Traps, 347
Tristate buffer, 74–75, *187*
  HDL for, 186–187
  multiplexer built using, 84–85, 91–93
True, 8, 20–22, 58–59, 70, 74, 111–112, 116, 129, 176, 180, 205
Truth tables, 20
  ALU decoder, *399*, *404*
  with don't cares, 69, 81–83, 205
  multiplexer, 83
  seven-segment display decoder, 79
  SR latch, 111, 112
  with undefined and floating inputs, *187–188*
TSOP. *See* Thin small outline package (TSOP)
TTL. *See* Transistor-Transistor Logic (TTL)
Two's complement numbers, 16–18
Two-bit dynamic branch predictor, 460
Two-cycle latency of `LDR`, 435
Two-level logic, 69
`typedef`, 541.e31–541.e32

U

UART. *See* Universal Asynchronous Receiver Transmitter (UART)
Unconditional branches, 308, *309*
Undefined instruction exception, 347
Unicode, *315*
Unit under test (UUT), 220
Unity gain points, 24
Universal Asynchronous Receiver Transmitter (UART), 531.e17–531.e23
  hardware handshaking, 531.e18
Universal Serial Bus (USB), 270, 531.e18, 531.e59
  USB 1.0, 531.e59
  USB 2.0, 531.e59
  USB 3.0, 531.e59
Unsigned multiplier, *217*, 252–253
Unsigned numbers, 18
Upton, Eben, 531.e4
USB. *See* Universal Serial Bus (USB)
USB links, 531.e63–531.e64
  FTDI, 531.e63
    UM232H module, *531.e64*
Use bit (U), 502

V

Valid bit (V), 496
Variables in C, 541.e7–541.e11
  global and local, 541.e9–541.e10
  initializing, 541.e11
  primitive data types, 541.e8–541.e9
V_{CC}, *23*. *See also* Supply voltage, V_{DD}
V_{DD}, 22, *23*. *See also* Supply voltage
Vector processor, *460*
Verilog. *See* SystemVerilog
Very High Speed Integrated Circuits (VHSIC), 175. *See also* VHSIC Hardware Description Language (VHDL)
VGA (Video Graphics Array) monitor, 531.e36–531.e42
  connector pinout, *531.e37*
  driver for, *531.e39–531.e42*
VHDL. *See* VHSIC Hardware Description Language (VHDL)
VHSIC. *See* Very High Speed Integrated Circuits (VHSIC)
VHSIC Hardware Description Language (VHDL), 173–175
  accessing parts of busses, 188, 192
  bad synchronizer with blocking assignments, 209
  bit swizzling, 188
  blocking and nonblocking assignment, 199–200, 205–208
  `case` statements, 201–202, 205
  combinational logic using, 177–193, 198–208, 217–220
  comments, 180
  conditional assignment, 181–182
  data types, 213–217
  decoders, 202–203, 219
  delays (in simulation), 189
  divide-by-3 FSM, 210–211
  finite state machines (FSMs), 209–213
    Mealy FSM, 213
    Moore FSM, 210, 212
  full adder, 184
    using always/process, 200
    using nonblocking assignments, 208
  history of, 175
  `if` statements, 202
  internal signals, 182–184
  inverters, 178, 199
  latches, 198
  logic gates, 177–179
  multiplexer, 181–183, 190–193, 218–219
  multiplier, 217
  numbers, 185–186
  operators, *185*
  parameterized modules, 217–220
    N:2^N decoder, 219
    N-bit multiplexers, 218, *219*
    N-input AND gate, 220, *220*

- priority circuit, 204
  - reduction operators, 180–181
  - using don't cares, *205*
- reduction operators, 180–181
- registers, 193–197
  - enabled, 196
  - resettable, 194–196
- sequential logic using, 193–198, 209–213
- seven-segment display decoder, 201
- simulation and synthesis, 175–177
- structural models, 190–193
- synchronizer, 197
- testbench, 220–224
  - self-checking, 222
  - simple, 221
  - with test vector file, 223–224
- tristate buffer, 187
- truth tables with undefined and floating inputs, 187, *188*
- z's and x's, 186–188, 205

Video Graphics Array (VGA). *See* VGA (Video Graphics Array) monitor

Virtual address, 509
- space, 515

Virtual memory, 490, 508–518
- address translation, 509–512
- cache terms comparison, 510
- memory protection, 515
- multilevel page tables, 516–518
- page fault, 509–510
- page number, 511
- page offset, 511
- pages, 509
- page table, 512–513
- replacement policies, 516
- translation lookaside buffer (TLB), 514–515
- write policy, 506–507

Virtual page number (VPN), 512

Virtual pages, 509

V_{SS}, *23*

W

Wafers, 28

Wait for event (`WFE`) instruction, 358

Wait for interrupt (`WFI`) instruction, 358

Wall, Larry, *20*

WAR hazard. *See* Write after read (WAR) hazard

WAW hazard. *See* Write after write (WAW) hazard

Weak pull-up, 33

Weird number, 18

`WFE`. *See* Wait for event (`WFE`) instruction

`WFI`. *See* Wait for interrupt (`WFI`) instruction

`while` loops, 312, 541.e19

White space, 180

Whitmore, Georgiana, 7

Wi-Fi, 531.e61

Wilson, Sophie, *472*

Wire, 67

Wireless communication, Bluetooth, 531.e42–531.e43

Wordline, 264

Write after read (WAR) hazard, 464. *See also* Hazards

Write after write (WAW) hazard, 464–465

Write policy, 506–507
- write-back, 506–507
- write-through, 506–507

X

X. *See* Contention (x); Don't care (X)

x86
- architecture, 360–368, *362*
  - big picture, 368
  - branch conditions, *366*
  - instruction encoding, 364–367
  - instructions, 364–367
  - memory addressing modes, *363*
  - operands, 362–363
  - peculiarities, 368
  - registers, 362
  - status flags, 363

Xilinx FPGA, 275

XNOR gate, 21–22

XOR gate, 21

Z

Z. *See* Floating (Z)